Rapport de Stage de DEA

Spécialité Informatique, Université d'Evry-val-d'Essonne

Sécurisation des primitives de communication bas niveau dans une machine parallèle de type grappe de PC

Emmanuel Dreyfus

Stage effectué au Laboratoire d'Informatique de l'Université de Paris 6 (LIP6)
4 place Jussieu, 75252 Paris Cedex 05, sous l'encadrement du Pr. Alain Greiner
(Mars 2000 - Septembre 2000)

Résumé

La machine MPC est une machine parallèle de type grappe de PC. Elle a été développée avec la conviction que le temps entre deux erreurs matérielles était de l'ordre de la dizaine d'années. A cause de problèmes matériels imprévus, ce temps entre deux fautes est en réalité de l'ordre de 2 minutes, dans le pire des cas. Des mécanismes de reprise sur erreur, qui paraissaient à l'origine inutiles, sont donc indispensable. Dans un premier temps, une sécurisation logicielle des primitives de communication haut niveau a été mise en œuvre. Mais cette solution n'est pas pleinement satisfaisante, car on ne peut utiliser les primitives bas niveau, qui permettent les plus hautes performances, de façon sure.

Une nouvelle version du contrôleur réseau de la machine MPC est actuellement en cours d'élaboration, et ce nouveau circuit sera reprogrammable. Le but de mon stage a été d'étudier le problème de la sécurisation de la primitive bas niveau de la machine MPC, en supposant que l'on dispose d'un contrôleur réseau reprogrammable. Il s'agissait donc d'élaborer un mécanisme de sécurisation et de définir ce qui devait être implémenté materiellement et ce qui devait être implémente logiciellement, pour pouvoir fournir une primitive de communication bas niveau sécurisée.

Abstract

The MPC project aim is to provide a low cost parallel computer, based on networked industry standard PCs. During the MPC first run developpement, it was the assumed that the mean time between failures was about ten years. But in practice, because of unexpected hardware failures, it is lowered down to about a couple of minutes. A software-based error recovery protocol was developped to address this issue, but it only secured the higher MPC communication layer. The lower MPC comunication layer, which enable the highest performance, was left unsecured.

A new version of the MPC network controler is currently being developped, and this new version will be a programmable chip. The aim of my training period was to propose a security feature for the lower MPC communication layer, using the new network controler. This job included deciding what tasks were to be done in hardware, and what tasks were to be done in software.

Remerciements

Je remercie l'ensemble de l'équipe de l' ASIM pour m'avoir accueilli dans leur département. En particulier, je remercie Alain Greiner, Franck Wajsbürt, Alexandre Fenyö, et Jean-Lou Desbarbieux pour l'aide qu'ils ont pu m'apporter dans la mise au point de mes travaux. Enfin, je remercie Daniel Millot, de l'Institut National des Télécommunications, pour la relecture du présent rapport.

Table des matières

Résumé 3

Remerciements 4

Tables des matières 5

I-Introduction et contexte du stage 7
I.2-Le projet MPC 7
I.3-Le sujet du stage 8
I.3-Le contenu de ce document 9

II-Architecture générale de la machine MPC/1 10
II.1-Objectif : zéro copie 10
II.2-Aspects matériels 11
II.3-Exploitation logicielle des cartes FastHSL 13
II.4-Vocabulaire 13
II.5-Fonctionnement global de la primitive PUT 14
II.6-Primitives de communication de plus haut niveau : SLR 16
II.7-Limitations de la machine MPC/1 18
II.7.a-MTBF du lien HSL plus faible que prévu 18
II.7.b-Accès en configuration sur jeu de composant 440BX 19
II.7.c-Fonctionnalités supplémentaires 19

III Sécurisation des communications dans la machine MPC/1 20
III.1-Rappel des contraintes existantes 20
III.2-Sécurisation au niveau PUT ou SLR? 20
III.3-Le protocole SCP/P 21

IV-La machine MPC/2 23
IV.1-Un nouveau contrôleur réseau: ANI 23
IV.2-Co-conception matérielle/logicielle 23
IV.3-Sécurisation au niveau PUT ou SLR? 24

V-Sécurisation au niveau PUT dans MPC/2 25
V.1-Détection des pertes de paquets 25
V.2-Accusés de non-réceptions ou accusés de réception? 25
V.3-Perte d'un paquet 26
V.4-Perte ou retard d'un accusé de réception 28
V.5-Condition d'envoi d'un accusé de réception. 29
V.6-Quelques cas pathologiques 32

V.6.a-Une émission entre un acquittement perdu et la retransmission correspondante 32
V.6.b-Nœud destinataire inaccessible 33
V.6.c-Congestion 33
V.6.d-Adaptativité 33
V.7-Impact sur les performances 34
V.7.a-Latence 34
V.7.b-Débit 36
V.8-sécurisation au niveau SLR 38

VI-Proposition d'implémentation de S/PUT-1.0 40
VI.1-Implémentation des renvois: la table des messages en transit 40
VI.1.a-Table des messages en transit indexée par le temps 41
VI.1.b-Table des messages en transit avec champ d'échéance 43
VI.2-Implémentation du rejet des doublons: les numéros de séquence 48
VI.3-Détection de la réception complète des paquets d'un messages bis 50
VI.4-Implémentation du mécanisme d'envoi des accusés de réception 51
VI.5-Bilan des contraintes imposées par la sécurisation 52
VI.6-S/PUT-1.0 en bref 53
VI.6.a-Présentation des structures de données 53
VI.6.b-Présentation des différents processus 54
VI.6.c-Opérations sur la table des messages en transit 55
VI.6.d-Comportement du processus d'émission 57
VI.6.e-Comportement du processus de surveillance des messages non acquittés 60
VI.6.f-Comportement du processus de réception 61
VI.6.g-Comportement du processus d'envoi des acquittements 62
VI.6.h-Formats des paquets 63

VII-Performances 64
VII.a-Latence 64
VII.b-Débit 64
VII.c-MTBF 65

VIII-Conclusions 66

Références 67

Annexe : des idées en vrac 68

I-Introduction et contexte du stage

I.I-Le projet MPC

Cette partie fait une brève introduction au enjeux du parallélisme, et explique les motivations du projet MPC.

Depuis la fin des années 60, les circuits intégrés des systèmes informatiques évoluent selon la loi de Moore, c'est à dire en suivant un doublement du nombre de transistors intégrés sur une surface donnée tous les 18 mois, pour un prix restant inchangé. Aux premiers jours, ces progrès en miniaturisation des circuits intégrés permettaient d'obtenir un doublement de performances sans augmentation du coût, sur une échelle de 18 mois. Grossièrement, plus un circuit est miniaturisé, moins il dissipe de chaleur, et plus on peut lui imposer une haute fréquence de fonctionnement.

Cependant, cette évolution a des limites à cause des difficultés d'intégration de circuits de plus en plus miniaturisés. La loi de Moore a ainsi commencé a subir une inflexion en 1992. De nos jours, pour continuer à augmenter la puissance des machines, on doit ajouter le parallélisme à la miniaturisation. L'introduction du parallélisme permet d'utiliser plusieurs circuits d'une puissance donnée en même temps, afin d'obtenir une puissance de calcul supérieure à ce qu'un seul circuit aurait pu faire. Dans les micro-ordinateurs, on retrouve le parallélisme un peu partout : pipelines et unités de calcul multiples dans les processeurs, contrôleurs ou coprocesseurs déchargeants le processeur de certaines tâches, systèmes multiprocesseurs, etc...

Pour obtenir plus de parallélisme, les approches sont nombreuses. Parmi elles, la machine parallèle de type "grappe de PC" est assez populaire aujourd'hui. Dans de tels systèmes, on utilise des compatibles PC, qui sont des machines peu chères, comme nœud de calcul. Chaque nœud possède sa propre mémoire, et exécute ses instructions sur ses données propres. On est en présence d'un système Multi-Instruction Multi-Données à Mémoire Distribuée (MIMD-MD)

Ce choix d'architecture permet d'éviter l'utilisation de machines dédiées coûteuses : une machine Single-Instruction Multi-Données (SIMD) requiert un processeur spécifique (donc cher), un système à mémoire partagée demande des mécanismes complexes de gestion de la cohérence des caches et de concurrence d'accès à la mémoire. Utiliser un réseau de PC assure donc un coût réduit.

Mais ce choix présente aussi un inconvénient majeur : les PC ne sont pas fais pour faire du calcul parallèle. Si du côté du processeur, grâce à la forte demande en puissance de calcul pour les jeux vidéo, on a des performances tout à fait correctes, du côté des communications entre processeurs, les moyens peu chers (réseaux ethernet, token ring...) dont on dispose sur un PC ne sont pas à la hauteur.

C'est sur ces constatations que se positionne le projet MPC : d'une part, aujourd'hui, le processeur d'un PC a des performances permettant de concurrencer un nœud de calcul d'une grosse machine parallèle. D'autre part, l'une des plus fortes limitations aujourd'hui dans les machines parallèles de type "grappes de PC", c'est le manque de performances des moyens de communications.

Dans le cadre de l'activité de design de circuits intégrés du département, un circuit routeur pour réseau hautes performances, nommé R-Cube a été développé. Ce circuit se destine a plusieurs applications, telles que les commutateurs gigabit ethernet ou dans le cas qui nous intéresse, les machines parallèles.

Le projet MPC regroupe des chercheurs du département ASIM du Laboratoire d'Informatique de l'Université de Paris 6 (LIP6), du PRiSM de l'Université de Versailles Saint Quentin (UVSQ), du LARIA de l'Université de Picardie Jules Verne, et des départements informatiques de l'Institut National des Télécommunications d'Evry (INT-Evry) et de l'Ecole Nationale Supérieure de Télécommunications de Paris (ENST-Paris).

Le but du projet est de développer une machine parallèle à coût réduit, en utilisant du matériel standard sur le marché, à savoir des compatibles PC, auxquels on fournit un réseau hautes performances. Les compétences mises en œuvre vont du design de circuits intégrés au développement de composants logiciels pour exploiter les circuits.

I.2-Le sujet du stage

Mon stage au LIP6 comportait deux parties clairement distinctes : la première était un travail d'ingénierie, faisant office de stage de fin d'études pour la dernière année du cycle ingénieur INT. Elle consistait à travailler sur la version Linux des composants logiciels exploitant les cartes réseaux de la machine MPC de première génération (MPC/1). Cette travail a donné lieu à un rapport de stage [ED] et à une soutenance, à l'INT.

Outre un perfectionnement dans les aspects systèmes, la partie ingénieur du stage m'a permis d'examiner assez profondément le fonctionnement de la machine MPC/1, ce qui m'a été fort utile pour la deuxième partie du stage. Cette deuxième partie correspond au travail de DEA, dont le présent rapport rend compte. Le sujet en est la sécurisation des primitives de communications bas niveaux dans la machine MPC.

MPC/1 a été conçu dans la croyance d'une fiabilité extrêmement élevée du lien (un temps entre deux erreurs de l'ordre du milliard de secondes, en théorie). L'expérience nous a révélé que cette supposition était erronée. Différents problèmes au niveau des connecteurs des câbles et au niveau du circuit R-Cube lui même font que le temps entre deux fautes n'est que de l'ordre de quelques heures. Les fautes elles mêmes sont assez mineures (perte d'un ou deux paquets), mais comme MPC/1 a été conçue à l'origine sans mécanismes de reprise sur erreur, l'utilisation de la machine est fortement compromise par ce problème.

Des travaux ont été réalisés par Alexandre Fenyö, au cours de sa thèse, pour corriger ces problèmes, et assurer la sécurisation des communications dans MPC/1 [AF]. Les composants matériels de MPC/1 n'étant pas reprogrammables, cette sécurisation ne pouvait être que logicielle. On ne pouvait que faire avec le matériel et trouver un moyen d'obtenir une communication sécurisée. Le résultat de ce travail est SCP, un protocole implémenté logiciellement, sécurisant l'API (*Application programming Interface*) SLR, qui corresponds à la primitive de communication de haut niveau de la machine MPC/1.

La sécurisation des communications de MPC/1 présente une note pessimiste: il n'est pas possible de sécuriser les communications au niveau le plus bas, à savoir au niveau de la primitive PUT. Aujourd'hui, la deuxième génération de la machine MPC (MPC/2) est en cours d'élaboration. Le sujet de mon stage a donc été d'examiner quels seraient les besoins matériels et/ou logiciels dans la machine MPC/2 pour pouvoir assurer la sécurisation des communications au niveau le plus bas.

I.3-Le contenu de ce document

Le présent mémoire rapporte les travaux accomplis au cours de mon stage de DEA sur le projet MPC. Pour cela, il commence, dans la partie II, par introduire la machine MPC de première génération (MPC/1), en étudiant avec un niveau de détail relativement élevé l'architecture de la machine et son fonctionnement. Cette partie fournit les bases nécessaire à la compréhension de la suite du rapport. Les lecteurs familiers avec le projet MPC peuvent donc commencer directement leur lecture à la partie III.

La partie III fait le bilan sur les problèmes de sécurisation de la machine MPC/1. Après avoir situé les contraintes dues à l'architecture de la machine MPC/1, elle décrit les possibilités de sécurisations, ainsi que la solution proposée par Alexandre Fenyö pour sécuriser les couches de communication de haut niveau.

On décrit ensuite, dans la partie IV, les caractéristiques de la machine MPC nouvelle génération (MPC/2), c'est à dire essentiellement les possibilités offertes par le nouveau contrôleur réseau actuellement en cours de développement au LIP6.

Les parties V, VI, et VII correspondent au travail réalisé pendant mon stage de DEA. La partie V aborde le problème de la sécurisation sur MPC/2, et dégage les besoins à satisfaire et les mécanismes requis pour obtenir une sécurisation sur la machine MPC/2. Au cours de cette partie, les différentes fautes pouvant survenir sont examinées, et des solutions pour corriger ces erreurs sont proposées.

La partie VI est une proposition d'implémentation d'une primitive de communication bas niveau sécurisée: S/PUT. Elle reprends les besoins dégagés dans la partie V, et définit comment les choses pourraient être implémentées pour obtenir la sécurisation. La répartition des tâches entre matériel et logiciel est traitée au cours de cette partie. La partie VI, se finit par une référence de S/PUT, qui synthétise les différents mécanismes et comportements qui ont été décrits jusque là.

Les performances théoriques de S/PUT sont finalement étudiées de façon sommaire dans la partie VII. On y calcule des ordres de grandeur de la latence et du débit disponibles avec cette nouvelle primitive de communication.

Il est supposé dans ce rapport que le lecteur dispose de connaissances générales en informatique, et plus particulièrement en architecture et programmation des systèmes parallèles.

II-Architecture générale de la machine MPC/1

Cette partie est consacrée à la description du fonctionnement général de la machine MPC. Elle décrit les motivations des choix de fonctionnement (le mode zéro copie), la façon dont les données sont transférées, ainsi que la façon dont le logiciel gère tout cela.

II.1-Objectif : zéro copie

Les problèmes de recopie mémoire sont parmi ceux qui nuisent le plus aux performances des communications entre machines dans un réseau de PC. La Fig. 1 illustre les différentes recopies qui se produisent lors d'un échange quelconque sur un réseau classique.

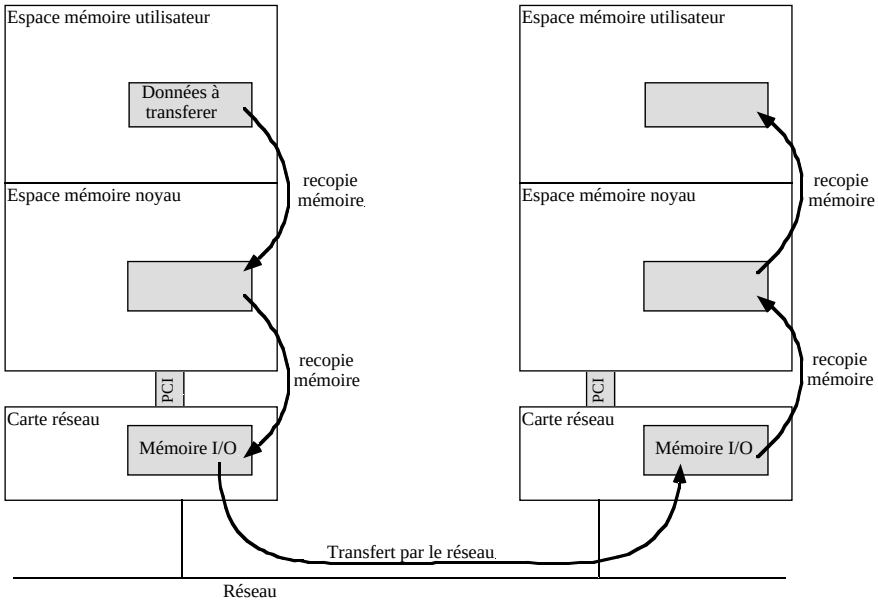

Fig 1: Différentes recopies mémoires mises en jeu lors d'un transfert de paquet sur un réseau classique (par exemple ethernet). On a parfois des recopies supplémentaires pour l'empaquetage.

Typiquement, un processus utilisateur utilise une primitive d'envoi, un *send()*. Cette primitive est implémentée par un appel système ou par un appel de fonction dans une librairie. Dans ce dernier cas, la librairie effectue quelques traitements, et finit par faire un appel système. On en revient donc toujours à un appel système, c'est à dire à un passage en mode noyau. Ce cheminement est inévitable, car pour des raisons de sécurité et de partage des ressources, seul le noyau peut avoir accès au matériel.

Au cours de l'appel système, le processeur va recopier les données à transmettre depuis l'espace mémoire du processus utilisateur vers l'espace mémoire du noyau. Une fois en mode noyau, c'est le pilote de la carte réseau qui travaille. Il va généralement recopier les données à transmettre dans une mémoire d'entrée/sortie situé sur la carte. Une fois les données copiées dans cette mémoire d'entrée/sortie, c'est au matériel de s'occuper de leur acheminement sur le réseau.

Les données arrivent alors sur la machine réceptrice, dans la mémoire d'entrée sortie de la carte réseau. Quand un paquet de donnée est reçu en entier, la carte va lever une interruption pour signaler au pro-

cesseur de la machine destination que des données sont arrivées. Cette interruption fait passer le processeur de la machine réceptrice en mode noyau, dans le pilote de la carte réseau. Les données vont alors être recopiées de la mémoire d'entrée/sortie de la carte vers un tampon dans l'espace mémoire du noyau.

Après la réception des données, différents traitements sont possibles. Si les données s'avèrent être à destination d'un processus ayant requis des communication asynchrones, il faut par exemple lui notifier l'arrivée des données par un signal (typiquement un *SIGIO*). Si le processus destinataire a déjà exécuté une primitive de réception, un *receive()*, il avait été endormi par un *sleep()* en attendant l'arrivée des données et il faut le réveiller par un *wakeup()*. Enfin si le processus destinataire n'a rien demandé, il faut conserver les données dans l'espace mémoire noyau en attendant qu'il veuille bien les demander par un appel à une primitive de réception de données.

Une fois que le processus destinataire est prêt à recevoir les données, le processeur doit finalement copier l'ensemble des données pour les ramener de l'espace mémoire noyau vers l'espace mémoire utilisateur.

Durant cette communication toute simple, les processeurs des machines source et destination ont chacun recopié 2 fois les mêmes données. Ceci représente un temps considérable si de grosses quantités de données sont échangées. Et en tout état de cause, ceci représente du temps perdu pour le calcul. Le circuit routeur R-Cube est capable de débiter 1Gb/s. C'est un débit assez considérable, mais si ce circuit est exploité avec les mécanismes classiques décrits ci dessus, on comprend que les performances ne seront pas au rendez-vous, car les processeurs seront essentiellement occupés à copier des données.

L'objectif dans la machine MPC est donc d'implémenter des mécanismes de communication présentant la propriété suivante : les données à transférer ne doivent jamais être copiées par le processeur. On parle de mode zéro copie.

Pour arriver à cet objectif, on va faire appel à la DMA (*Direct Memory Access*). La DMA est un mécanisme permettant à un périphérique d'avoir un accès direct en lecture ou en écriture à la mémoire centrale de l'ordinateur, sans passer par le processeur.

On a donc adjoint à R-Cube un second composant matériel, nommé PCIDDC, implémentant une primitive d'écriture dans la mémoire centrale d'un nœud de calcul distant (protocole DDSLRP : *Direct Deposit StateLess Receiver Protocol*). La DMA est utilisée au départ et à l'arrivée pour éviter les recopies mémoires par le processeur. PCIDDC réalise l'intermédiaire entre le bus PCI de la machine hôte et le routeur R-Cube. Les couches logicielles n'ont qu'à indiquer à PICDDC l'adresse source et la longueur du bloc à transférer, ainsi que l'adresse destination où il doit être écrit sur le nœud distant.

Plus d'informations sur R-Cube [R3] et PCIDDC [DDC] sont disponibles sur le site Internet de MPC.

Un tel mécanisme permet de tirer partie des possibilités offertes par un routeur haute performance tel que R-Cube, tout en conservant un maximum de temps pour le calcul sur les nœuds.

II.2-Aspects matériels

Les circuits R-Cube et PCIDDC sont montés sur une carte PCI, appelée carte FastHSL, qui prend place dans un slot d'extension du PC. Suivant sa révision, la carte FastHSL comprend 7 connecteurs Harting ou 8 connecteurs Lemo, sur lesquels on peut brancher des câbles bi-coaxiaux. Du fait du nombre de connecteurs disponibles, il est possible de réaliser des réseaux fortement connexes pour des machines parallèles de 7 nœuds de calcul.

Pour les cartes à 8 connecteurs, seuls 7 connecteurs sont utilisables lors de l'utilisation dans un PC. En effet, le routeur R-Cube possède 8 interfaces, dont une est utilisée pour la communication avec PCIDDC. Le huitième connecteur permet l'utilisation de R-Cube comme nœud flottant, c'est à dire non raccordé à un PC. Cette possibilité n'a pas été actuellement exploitée.

Le circuit PCIDDC original (PCIDDC first run) comptait un certain nombre d'erreurs. Certaines d'entre elles ont été corrigées dans PCIDDC second run. Parmi les cartes à connecteurs Harting, certaines sont en PCIDDC first run, et d'autres en PCIDDC second run. Toutes les cartes à connecteur Lemo utilisent des circuits PCIDDC second run. Pour s'y retrouver, on parle de révisions de cartes FastHSL :

Rev. A PCIDDC first run, 7 connecteurs Harting
Rev. B PCIDDC second run, 7 connecteurs Harting
Rev. C PCIDDC second run, 8 connecteurs Lemo

Chaque nœud de calcul de la machine MPC est équipé d'une carte FastHSL. Les cartes FastHSL sont liées entre elles par le réseau de données, composé de liens HSL. Par ailleurs, pour les opérations de configuration, la machine MPC requiert un réseau de contrôle. Celui ci peut être n'importe quel réseau capable de transporter le protocole TCP/IP. Actuellement, c'est un réseau ethernet qui est employé. La Fig. 2 donne un aperçu général d'une machine MPC à 4 nœuds.

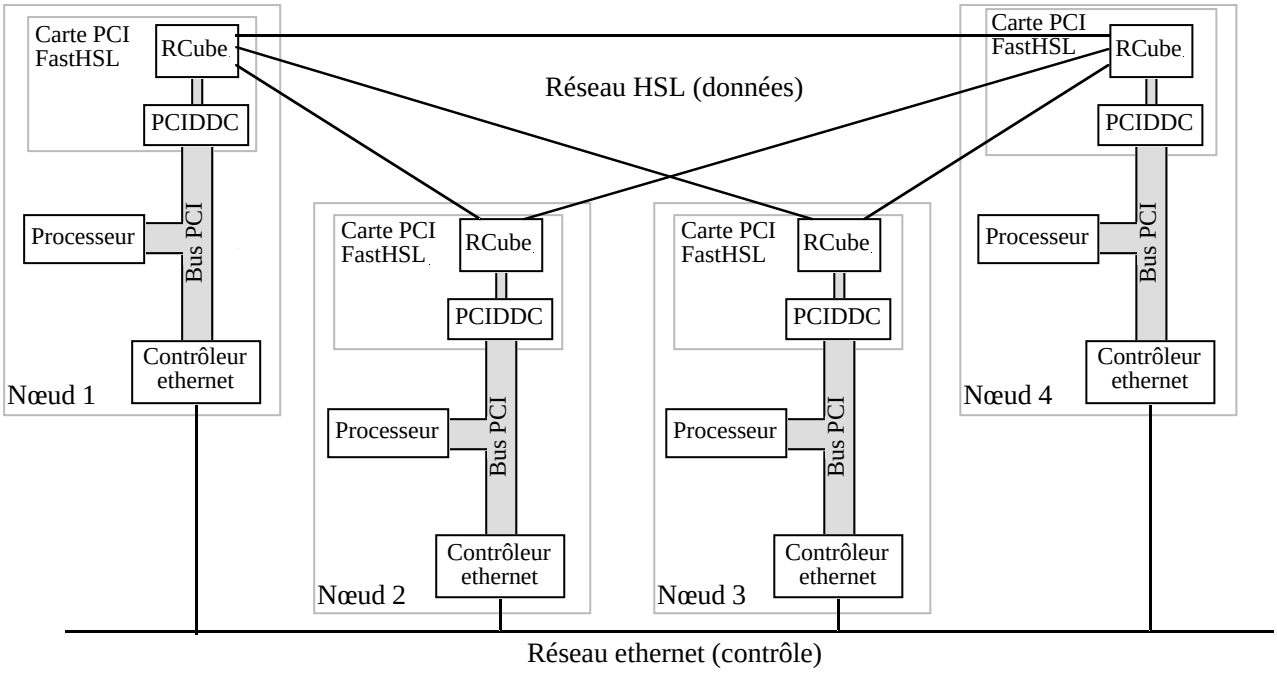

Fig. 2 : Machine MPC à 4 nœuds. On a choisi ici d'utiliser un réseau HSL fortement connexe.

II.3-Exploitation logicielle des cartes FastHSL

L'exploitation logicielle des cartes FastHSL est assurée par MPC-OS (pour *MPC Operating System*) [EMU]. MPC-OS est implémenté sous forme de modules à charger dynamiquement dans les noyaux des systèmes FreeBSD et Linux :

• Le module CMEM, qui est un gestionnaire de mémoire physiquement contiguë.

• Le module HSL, chargé de l'exploitation proprement dite de la carte, implémente un certain nombre de services pour les couches logicielles de plus haut niveau. Il implémente aussi l'interface de programmation PUT, qui permet l'écriture dans un nœud distant en utilisant la carte fastHSL. PUT est la primitive de communication de plus bas niveau dans la machine MPC.

Le module HSL doit dialoguer avec PCIDDC pour envoyer des pages de mémoire sur les nœuds distants. Le reste de cette partie aborde les moyens employés par le module HSL pour communiquer avec PCIDDC.

La norme PCI impose aux cartes PCI de fournir des registres de configuration. Ces registres sont accessibles en lecture et en écriture, et ils permettent les fonctions d'auto-détection et d'identification de la carte, ainsi que sa configuration lors du démarrage de la machine.

Habituellement, les registres de configuration ne sont utilisés qu'au démarrage de la machine pour configurer la carte. Lors de ces opérations, la mémoire d'entrée/sortie de la carte est remappée dans l'espace de mémoire virtuelle du noyau. Dans cet espace se trouvent des registres de contrôle et de statut qui servent à contrôler la carte lors du fonctionnement courant. On les manipule alors par accès mémoire.

La carte FastHSL présente une particularité dans son usage des registres de configuration : il n'y a pas de mémoire d'entée/sortie sur la carte, et les registre de statut et de commande de PCIDDC sont accessibles dans l'espace de configuration de la carte, et ceci durant le mode d'opération normal.

Au démarrage de la machine, le système d'exploitation remappe les espaces de configuration des cartes PCI dans la mémoire virtuelle du noyau. Il suffira donc d'aller lire et écrire à la bonne adresse pour communiquer avec PCIDDC. De même, des opérations sur le bus PCI lui-même seront nécessaires, et pour cela on agira de la même façon sur les registres d'état et de contrôle du bus PCI (ils sont eux aussi remappés en mémoire virtuelle noyau lors du démarrage du système d'exploitation).

II.4-Vocabulaire

La spécification de PCIDDC utilise un certain nombre de termes qu'il convient de définir précisément avant d'aborder plus en détail le fonctionnement de la machine MPC.

Paquet HSL: Unité de données transférée sur le réseau HSL. C'est à PCIDDC de former et d'extraire les paquets, qu'il envoie et reçoit de R-Cube.

Page réseau (ou page) : Une page réseau est un espace de mémoire virtuelle mappée de façon continue en mémoire physique, que l'on désire transférer. On peut gérer des pages réseaux correspondantes aux pages de la mémoire virtuelle (La mémoire virtuelle est gérée par pages de 4ko), mais cela n'est pas une obligation: une page réseau a une longueur arbitraire, entre 1 et 65536 octets. La seule contrainte est la continuité en mémoire physique. Lors du transfert, les pages sont découpés en un ou plusieurs paquets HSL.

Message : Un message est un ensemble de page, dont on souhaite se voir notifier le transfert en une seule fois. Par exemple, si on envoie une colonne de matrice, on aura une page par élément (puisqu'ils ne sont pas rangés de façon contiguë en mémoire), mais l'ensemble des pages constituera un message, et c'est le transfert du message et non pas des pages individuelles qui sera notifié.
Numéro de nœud : Chaque machine sur le réseau HSL possède un numéro de nœud. Les numéros de nœuds sont statiques et assignés à l'initialisation du réseau par MPC-OS.
Identifiant de message (MI : *Message identifier*) : Chaque message, dans le contexte d'une communication entre deux nœuds, possède un identifiant unique. Pour assurer l'unicité des MI, chaque processus utilisateur sur chaque nœud se voit attribuer un intervalle de MI par MPC-OS.
Message court (SM : *Short Messages*) : Pour transmettre des messages de moins de 8 octets, PCIDDC propose un mécanisme de messages courts, qui évitent un certain nombre d'étapes dans le transfert.

II.5-Fonctionnement global de la primitive PUT

PCIDDC maintient dans la mémoire centrale de l'ordinateur deux tables essentielles au fonctionnement de la primitive de plus bas niveau de MPC : PUT.
• la liste des pages à émettre (LPE : *List of Page to Emit*)
• la liste des messages reçus (LMI : *List of Message Identifiers*)

L'emplacement et la taille de ces deux tables en mémoire centrale est indiquée à PCIDDC par un accès en configuration. PCIDDC accède ensuite aux tables par DMA.

La table LPE contient des entrées de type `lpe_entry_t`, définies ainsi :

```
typedef struct _lpe_entry{
     u_short   page_length;    /* Longueur du bloc à transférer */
     u_short   routing_part;   /* Numéro du nœud destination */
     u_long    control;        /* Options diverses */
     caddr_t   PRSA;           /* Adresse destination (sur nœud cible) */
     caddr_t   PLSA;           /* Adresse source (sur nœud local) */
} lpe_entry_t;
```

Pour envoyer une page de mémoire sur un nœud distant, il suffit d'écrire l'entrée de la table LPE correspondante, et signaler à PCIDDC la présence de cette entrée par un accès en configuration. L'entrée de LPE contient l'adresse locale du bloc à transférer (`PLSA`), sa longueur (`page_length`), le numéro de nœud sur lequel il faut le transférer (`routing_part`), et l'adresse à laquelle le bloc doit être transféré sur le nœud distant (`PRSA`).

Le champ `control` permet de déterminer certains comportements optionnels, comme par exemple l'envoi d'un message court. Dans ce cas, les champs `PRSA` et `PLSA` contiennent les données à transférer, au lieu d'indiquer leurs adresses source et destination.

La liste des messages reçus (LMI), est mise à jour par PCIDDC dès que toutes les pages d'un message ont été écrites dans la mémoire. Ses entrées sont de type `lmi_entry_t`, définies ainsi :

```
typedef struct _lmi_entry{
     u_short   packet_number; /* Identifiant du message (MI) */
     u_short   r3_statut;     /* registre R3 */
     u_short   reserved;      /* Reserved for future use */
```

```
    u_short    control;        /* Options diverses */
    u_long     data1;          /* Adresse source */
    u_long     data2;          /* Adresse destination */
} lmi_entry_t;
```

Le champ `packet_number` contient le MI du message.

Le champ `r3_statut` contient la valeur du registre R3 de PCIDDC, qui indique l'état des 8 liens de R-Cube.

Le champ `control` correspond au champ `control` de l'entrée de LPE associé au message.

Enfin, les champs `data1` et `data2` correspondent aux adresses sources et destination du bloc transféré, comme les champs `PRSA` et `PLSA` de l'entrée de LPE. Dans le cas des messages courts, ces champs contiennent les données qu'on a mis dans les champs `PRSA` et `PLSA` de l'entrée de LPE.

PCIDDC transfère toutes les données par DMA, donc de façon totalement asynchrone vis-à-vis du processeur. Il est donc nécessaire de pouvoir détecter la complétion des différentes tâches effectuées par PCIDDC. Ceci peut se faire de deux façons :

• par scrutation de la LPE et de la LMI

• en demandant à PCIDDC de générer une interruption lorsque la page correspondant à une entrée de LPE a été envoyée ou quand une entrée de LMI correspondant à une page arrivée, est ajoutée.

Chacune de ces deux méthodes présente son avantage : par scrutation, la latence d'un envoi est réduite, car elle ne comprend pas les interruptions. Par contre, le processeur doit être sollicité pour scruter périodiquement la LMI et la LPE. Ce temps de scrutation est ainsi perdu pour les calculs. La méthode par interruption permet d'éviter de perdre du temps à la scrutation, mais la latence des échanges en est allongée.

Les deux modes sont disponibles, mais pas simultanément. On choisit l'une ou l'autre en compilant MPC-OS avec l'option `PUT_MODEL_INTERRUPT_DRIVEN` (pour les interruptions) ou `PUT_MODEL_POLLING` (pour la scrutation).

La Fig. 3 donne une vue d'ensemble des différentes étapes nécessaires dans la transmission d'un bloc de mémoire (un message constitué d'une seule page, pour simplifier) :

• Etape 1 : écriture de l'entrée de LPE correspondante au bloc que l'on souhaite transférer.

• Etape 2 : accès en configuration à PCIDDC pour lui signaler la présence d'une nouvelle entrée à traiter.

• Etape 3 : PCIDDC lit l'entrée de LPE par DMA.

• Etape 4 : PCIDDC lit la page de mémoire à transférer par DMA.

• Etape 5 : transfert des données à acheminer de PCIDDC à R-Cube. Si la page est de taille importante, elle sera fragmentée en plusieurs paquets HSL.

• Etape 6 : transfert des données d'un routeur R-Cube à l'autre, à travers le réseau HSL. Il serait possible durant cette étape de traverser des routeurs R-Cube intermédiaires.

• Etape 7 : transfert des données de R-Cube à PCIDDC, sur le nœud destination.

• Etape 8 : transfert des données de PCIDDC vers la mémoire centrale sur le nœud destination. Une fois de plus, la DMA est utilisée pour éviter les recopies mémoires

Les étapes 4 à 8 se déroulent en réalité en même temps : elles forment un pipeline.

Une fois l'étape 4 terminée, l'étape 9 peut éventuellement être faite sur le nœud émetteur:

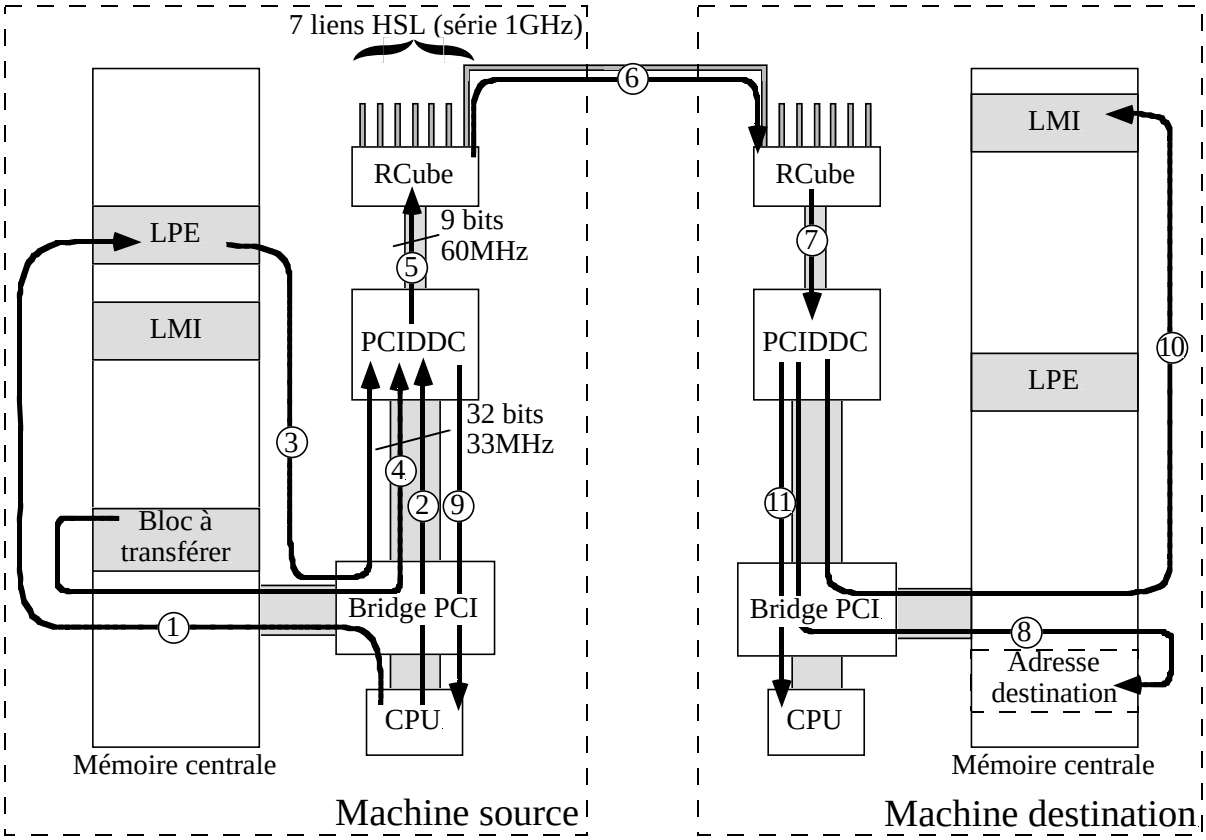

Fig. 3 : Différentes étapes mises en jeu lors du Transfert d'un bloc de mémoire par la machine MPC.

•Etape 9 (optionnelle) : signalisation de l'émission de la page par interruption.

Une fois l'étape 8 terminée, les étapes 10 et 11 peuvent être éventuellement faites sur le récepteur :
• Etape 10 (optionnelle) : mise à jour de la LMI par PCIDDC. Cette mise à jour est faite par une opération de DMA, au cours de laquelle l'entrée correspondant au message arrivé est ajoutée à la LMI.
• Etape 11 (optionnelle) : levée d'une interruption pour signaler à la machine destination qu'un message est arrivé.

L'activation des comportements optionnels est assuré par des drapeaux dans le champ *control*.

On peut trouver plus d'informations sur le fonctionnement de PUT dans le manuel du programmeur de MPC/1 [MPC]

II.6-Primitives de communication de plus haut niveau : SLR

Nous avons jusqu'ici illustré le fonctionnement de la machine lors de l'usage de la primitive de plus bas niveau de la machine MPC/1, à savoir PUT. MPC-OS implémente logiciellement des primitives de plus haut niveau: SLR/P et SLR/V. Cette partie décrit rapidement ces deux primitives.

PUT est une interface de très bas niveau : elle ne permet que d'écrire dans la mémoire d'un nœud distant. Il reste à la charge de l'utilisateur de PUT de s'arranger pour que l'émetteur place ses données dans un

tampon adéquat sur le récepteur. Du côté du récepteur, l'utilisateur doit se débrouiller pour détecter la complétion du transfert. Développer une application parallèle basé sur PUT est donc loin d'être quelque chose de trivial.

Il est donc hautement souhaitable de disposer de deux primitives d'envoi et de réception de plus haut niveau : *send()* et *receive()*. De telles primitives permettent l'envoi de données sans avoir à se préoccuper sur le nœud émetteur de la localisation du tampon de réception sur le nœud destinataire. De plus, la détection de la réception de données est rendue triviale, puisqu'elle se fait en utilisant la primitive de réception.

SLR/P et SLR/V implémentent de telles primitives *send()* et *receive()*. Ils proposent également un système de canaux virtuels, qui permettent de partager facilement une carte FastHSL entre plusieurs processus différents. SLR/P s'appuie sur PUT pour proposer les primitives d'envoi et de réception avec des adresses physiques, et SLR/V s'appuie sur SLR/P pour proposer le même service, mais avec des adresses virtuelles.

Examinons sommairement le fonctionnement de SLR/P. Le problème à résoudre est le suivant : lorsque l'utilisateur fait un *send()*, nous connaissons l'emplacement des données à envoyer, mais nous n'avons pas idée de l'endroit où les déposer sur le nœud destination. Et quand l'utilisateur fait un *receive()*, on sait où déposer des données, on ne sait pas aller les chercher (la primitive de bas niveau, PUT, ne permet pas d'aller chercher les données, elle ne sait que envoyer). Il faut attendre que les données arrivent, suite à un PUT effectué sur l'émetteur.

Une approche simpliste serait d'implémenter le *send()* par un envoi de données dans un tampon de réception situé à une adresse bien connue sur le nœud destinataire, et ensuite laisser à la charge du récepteur le travail de recopier des données dans le tampon de destination. Cette approche n'est pas très bonne, car elle s'écarte du mode zéro copies qui est l'objectif dans la machine MPC.

On doit donc utiliser un mécanisme plus ingénieux. Lorsqu'un processus sur le récepteur fait un *receive()*, on va envoyer un message de contrôle à l'émetteur en utilisant PUT. Ce message de contrôle va indiquer à l'émetteur ou est le tampon de réception sur le nœud destination. Lorsqu'un processus fait un *send()* sur le nœud émetteur, deux cas se présentent :
• Soit un message de contrôle dû à un *receive()* a déjà été envoyé, auquel cas on envoie par PUT les données dans le tampon en réception dont on connaît l'adresse sur le nœud destinataire
• Soit on n'a pas encore reçu de message de contrôle du nœud destinataire indiquant l'emplacement du tampon de réception. Dans ce dernier cas, le *send()* est mis en attente, et c'est la réception d'un message de contrôle dû à un *receive()* qui va provoquer l'émission des données par PUT. La fig. 4 donne un aperçu d'un échange SLR/P.

En résumé, le transfert des données est fait par le nœud émetteur une fois qu'il a connaissance des emplacements du tampon d'émission et du tampon de réception. Ceci revient à la conjonction des deux conditions suivantes :
• Le nœud émetteur a eu un *send()*
• Le nœud émetteur a reçu le message de contrôle dû au *receive()* sur le nœud récepteur.

La fig. 4 illustre les deux échanges SLR possibles, suivant l'ordre du *receive()* et du *send()*.

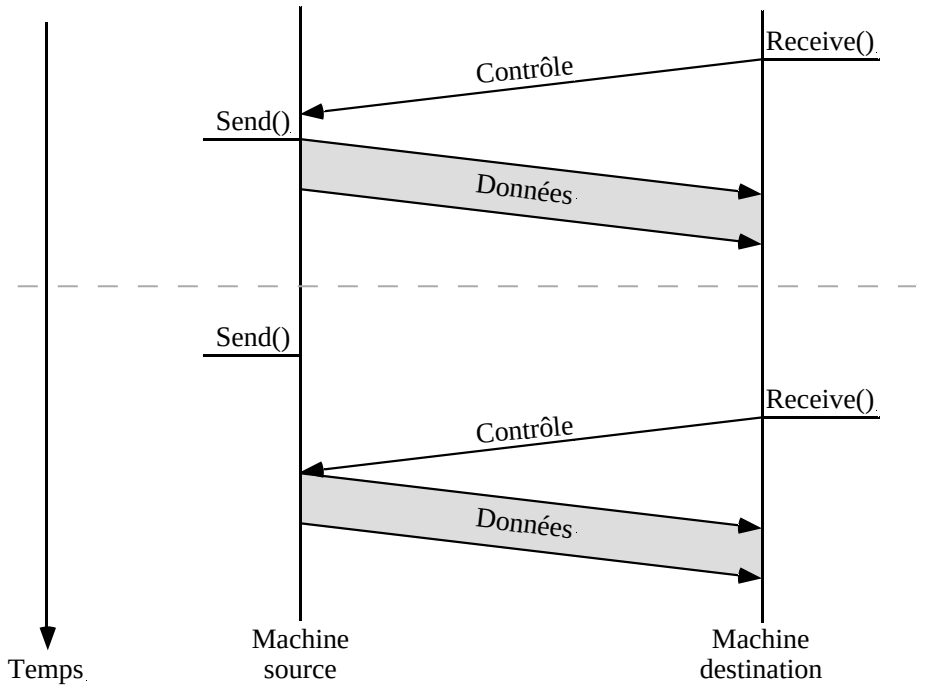

Fig. 4: Echanges SLR dans deux cas: en bas, le send() *a été fait avant le* receive(). *En haut le* receive() *a été fait avant le* send(). *Dans SLR, le* receive() *déclenche un message de contrôle indiquant où les données doivent être écrites. L'émetteur n'envoie les données qu'une fois qu'il a (1) reçu ce message de contrôle, et (2) eu un* send() *en attente.*

SLR/V fonctionne en utilisant SLR/P. La différence essentielle est que le message de contrôle n'a pas simplement a véhiculer l'adresse et la longueur d'un tampon, comme dans SLR/P, mais toute une liste de pages de mémoires virtuelles, le tampon de réception pouvant être fragmenté dans la mémoire physique.

SLR permet donc d'avoir une API assez générique de passage de message. Ceci rends le portage d'applications parallèles sur MPC/1 plus facile, mais a un coût important en termes de performances: la latence SLR est beaucoup plus importante que la latence au niveau PUT.

II.7-Limitations de la machine MPC/1

La machine MPC/1 connaît un certain nombre de problèmes. Examinons-les plus en détail.

II.7.a-MTBF du lien HSL plus faible que prévu

Comme cela a été évoqué au I.3, la machine MPC/1 a été conçue en imaginant que la fiabilité des liens HSL était extrêmement élevée. Les chiffres théoriques promettent un bit erroné tous les 10^{18} bits. Le lien HSL fonctionnant à 1Gb/s, ceci nous fait une erreur tous les milliard de secondes. Exprimé dans une unité plus raisonnable, cela nous fait un temps moyen entre deux fautes (MTBF: *Mean Time Between Failures*) de plus de 30 ans. Il est clair qu'avec une telle fiabilité, on peut se dispenser de gérer la récupération sur erreur. Un MTBF de 30 ans est très grand devant les MTBF de la plupart des autres constituants de la machine.

Hélas, le MTBF réel est bien plus faible que cette valeur attendue de 30 ans. En pratique, dans le pire cas, c'est à dire avec un réseau chargé au maximum, on subit des fautes avec un MTBF de deux minutes.

Ceci est principalement dû à un problème dans R-Cube qui rend le circuit particulièrement sensible à la gitte du front d'onde des paquets sur les liens HSL. Ce problème est accentué par l'usage des connecteurs Lemo, qui provoquent des couplages entre le lien montant et le lien descendant sur les câbles bi-coaxiaux utilisés pour relier les routeurs R-Cube.

Une erreur se manifeste par la perte d'un ou deux paquets sur le lien. Comme la machine MPC/1 n'a à l'origine aucun moyen de récupérer sur cette erreur, une faute provoque un arrêt de la machine. On en arrive donc à une situation où la reprise sur erreur n'est plus du tout inutile.

II.7.b-Accès en configuration sur jeu de composant 440BX

Autre problème: PCIDDC est fait pour être commandé par des accès en configuration. Habituellement, les cartes PCI sont commandées par des accès en configuration lors du démarrage de la machine, pour les initialiser. Une fois l'initialisation faite, sa mémoire est mappée dans la mémoire virtuelle de la machine, et on la commande par des accès mémoire. Les cartes PCI normales n'ont donc pas besoin d'accès en configuration à un rythme très élevé.

Dans le cas de PCIDDC, chaque envoi de page demandant un accès en configuration, on peut être amené à faire des accès en configuration à une fréquence extrêmement élevée, lorsque l'on envoie rapidement des petites pages (par exemple pour envoyer une colonne de matrice).

Ces accès en configuration posent problème lorsque la carte est utilisée avec des cartes mères utilisant le jeu de composants 440BX d'Intel. Un problème dans la passerelle PCI de ce jeu de composants provoque des échecs lorsque l'on fait des accès en configuration à un rythme trop soutenu. Par exemple, si on lit un registre d'état par accès en configuration avec une très haute fréquence, on finit par lire la valeur 0xFFFFFFFF. On sait que certains bits de ce registre sont reliés à la masse, donc on est sûr que ce comportement relève du dysfonctionnement.

Intel reconnaît d'ailleurs le problème, puisque le document indiquant la spécification du jeu de composant 440BX précise que les accès en configuration sont destinés à l'initialisation, et qu'il convient de faire attention si l'on s'en sert pour autre chose. Toutes les cartes PCI normales se trouvant sur le marché n'utilisant les accès en configuration que pour l'initialisation, le problème passe tout à fait inaperçu, sauf avec les cartes FastHSL.

II.7.c-Fonctionnalités supplémentaires

L'expérience acquise avec MPC/1, a permis de dégager des besoins qui n'étaient pas évidents lors de la conception de PCIDDC.

• Il manque à PCIDDC un mécanisme permettant de gérer au niveau matériel l'utilisation de la carte FastHSL par plusieurs processus. Ceci passe par l'utilisation de plusieures LPE, une par processus. Des expériences ont été faites dans ce sens avec RWU, développé dans le projet NOE [JLD]

• Les messages courts sont trop courts (!), il faudrait pouvoir en faire des plus longs. Les messages courts de MPC/1 font 8 octets, ce qui est une forte limitation.

• MPC/1 ne prévoit pas de mécanisme pour savoir de quel nœud vient un message. On est obligé d'utiliser une partie du champ MI à cet usage. De plus, le matériel de MPC/1 impose l'unicité des MI utilisés sur le réseau. Ces limitations sont assez contraignante pour le développement de protocoles basés sur PUT. On souhaite donc lever les contraintes imposées par le système sur le contenu du champ MI, afin que l'utilisateur puisse l'employer comme il l'entend. De plus, un champ MI plus étendu faciliterait le développement de protocoles de haut niveau.

III Sécurisation des communications dans la machine MPC/1

Cette partie aborde le problème de la sécurisation des communications sur MPC/1. Après un bref rappel des contraintes inhérentes à l'architecture de MPC/1, on y étudie les possibilités de sécurisation, en enfin la solution proposée pour sécuriser les communications de haut niveau.

III.1-Rappel des contraintes existantes

Rappelons brièvement les contraintes imposées par le mode zéro copies :

• Lorsque la fin d'un envoi est signalée, on ne peut plus utiliser le tampon d'émission. L'application a le droit de le modifier à loisir. Si on relit le tampon, on risque de ne plus avoir les données initiales
• Lorsque la fin d'une réception est signalée, on ne peut plus utiliser le tampon en réception. L'application a le droit de le modifier, et toute nouvelle écriture dans le tampon risque d'annuler des modifications faites par l'application.

et une contrainte imposée par PCIDDC :

• Il est impossible d'empêcher le dépôt d'une page sur le nœud récepteur.

III.2-Sécurisation au niveau PUT ou SLR?

Aucune tentative de sécurisation au niveau PUT n'a été tentée sur MPC/1. La raison est une question de facilité. En effet, au niveau PUT, la transaction commence par une demande d'émission chez l'émetteur, et finit par une signalisation d'arrivée de message sur le récepteur. Par contre, au niveau SLR, la transaction commence par l'appel à *receive()* sur le récepteur et se finit par la notification de réception (ou la complétion du *receive()*, si on a choisi un *receive()* bloquant) sur ce même nœud. Ceci suggère que l'émetteur peut se comporter de façon passive, et que c'est au récepteur de s'occuper de détecter les transaction non complétées.

Si l'on essayait de sécuriser PUT, on devrait provoquer l'envoi d'un accusé de réception à chaque fois que le récepteur reçoit un message. Dans cette voie, l'émetteur est chargé de vérifier la complétion des transactions.

On le voit, ces deux situations ne sont pas identiques. Dans le cas de la sécurisation de SLR, c'est au récepteur de veiller à la complétion des transactions, dans le cas de la sécurisation de PUT, c'est à l'émetteur.

La gestion de la complétion par l'émetteur pose un problème : comme le récepteur ne peut empêcher le dépôt d'une page, si l'émetteur ré-envoi une page alors que le tampon a été libéré sur le récepteur (ceci va arriver si un accusé se perd et que le récepteur croit la transaction terminée alors que l'émetteur croit le contraire), on va écraser des données appartenant à une application. D'autre part, si c'est le récepteur qui gère la complétion des transactions, ce problème disparaît : les données sont envoyées au récepteur à sa demande, et il ne peut donc pas y avoir écrasement des données contenues dans un tampon déjà libéré.

Le problème de la sécurisation de PUT sur MPC/1 n'est pas insoluble : il est probable qu'en ajoutant des acquittements d'acquittements, on parvienne à une solution acceptable. Mais pour des raisons de simplicité, c'est la sécurisation au niveau SLR qui a été choisie.

III.3-Le protocole SCP/P

Le développement de la sécurisation des communications dans la machine MPC/1 a été effectué par Alexandre Fenyö, au cours de sa thèse [AF]. Le protocole sécurisé remplaçant SLR/P s'appelle SCP/P. Il manipule des adresses physiques. Un protocole SCP/V s'appuie sur SCP/P pour fournir le même service avec des adresses virtuelles, et il se présente en remplaçant de SLR/V.

Avant de décrire le protocole SCP/P, il faut signaler une des limitations qu'il requiert pour fonctionner : le réseau ne doit pas être adaptatif. En effet, un des problèmes qu'il nous faut traiter est le retard d'un paquet. Le temps qu'un paquet peut passer dans le réseau n'est pas borné. En cas de retard d'un paquet, on ne peut donc pas attendre un certain temps, puis se dire que le paquet est forcément perdu.

Dans le cas d'un réseau non adaptatif, pour s'assurer qu'un paquet n'est pas resté bloqué dans le réseau, il suffit de faire un "ping-pong". Le comportement FIFO du réseau nous assure que si le ping-pong passe, alors aucun paquet n'est bloqué au milieu du réseau. Sur un réseau adaptatif, on perd cette garantie: le ping-pong peut emprunter une route différente de celle où un paquet reste bloqué, et faire un ping-pong ne permet donc pas de savoir si il reste des paquets bloqués dans le réseau.

Le protocole SCP/P se divise en plusieurs sous-protocoles :

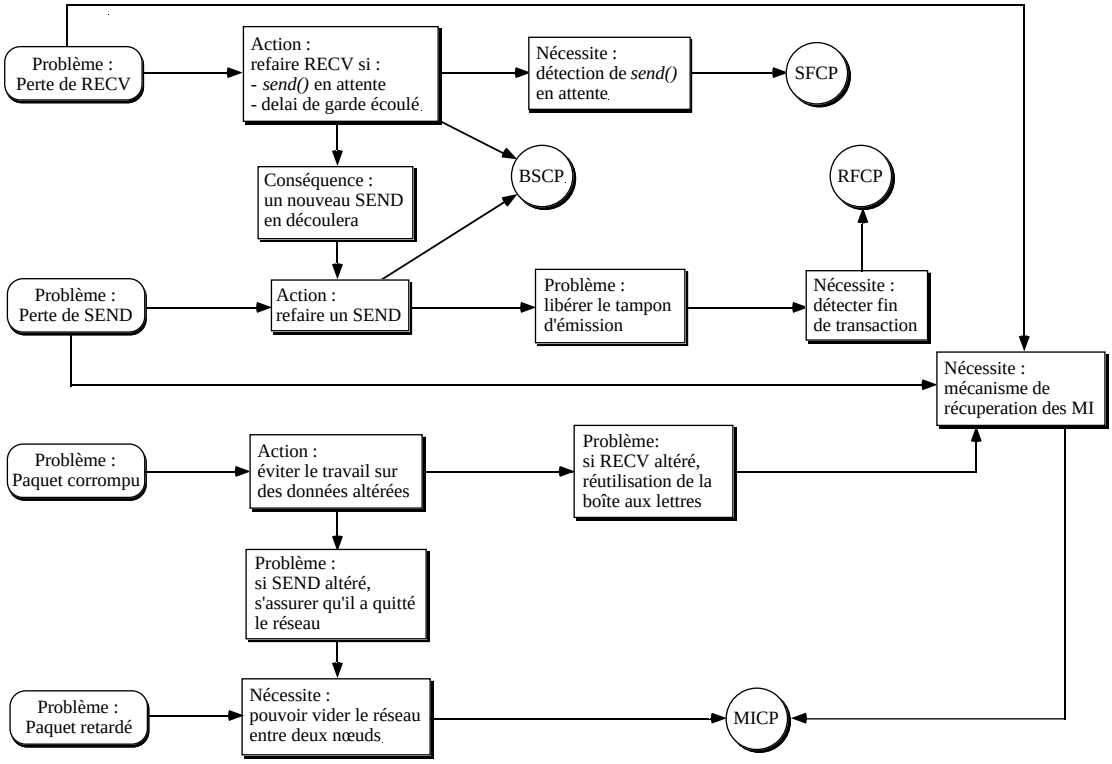

Fig. 5 : Construction du protocole SCP. Ce diagramme met en relief les problèmes pouvant survenir, les solutions à mettre en œuvre pour les régler, et le sous protocole assurant ces actions.

• SFCP permet de notifier le récepteur qu'un *send()* est en attente. Il s'agit d'un ping-pong, qui permet à

l'émetteur de s'assurer que le récepteur sait qu'un *send()* est en attente. En cas de non réponse, SFCP continuera à signaler le *send()* en attente, jusqu'à avoir un acquittement (le "pong" du poing-pong)
• RFCP permet au récepteur de signaler à l'émetteur que la transaction est achevée, et qu'il peut signaler la complétion du *send()*. De même qu'avec SFCP, il s'agit d'un ping-pong, et le récepteur signalera la fin de transaction jusqu'à avoir un acquittement
• BSCP est un sous-ensemble de SLR/P, qui comporte le *send()* et le *receive()*.
• MICP est un protocole permettant de vider le réseau entre deux nœuds suite à une faute, et de ré-utiliser les MI.

La figure 5 illustre les problèmes pouvant survenir dans une transaction, les besoins qu'ils impliquent pour pouvoir récupérer sur l'erreur, et le sous-protocole de SCP/P qui assure cela. La figure 6 illustre une transaction SCP/P sans faute. Le point important est que chaque étape est gardée par un accusé de réception, ce qui permet de détecter les pertes.

Le but de cette partie est de fournir un aperçu du protocole SCP/P, pas de l'expliquer dans le détail. Pour plus d'informations sur SCP/P, se reporter au rapport de thèse d'Alexandre Fenyö [AF].

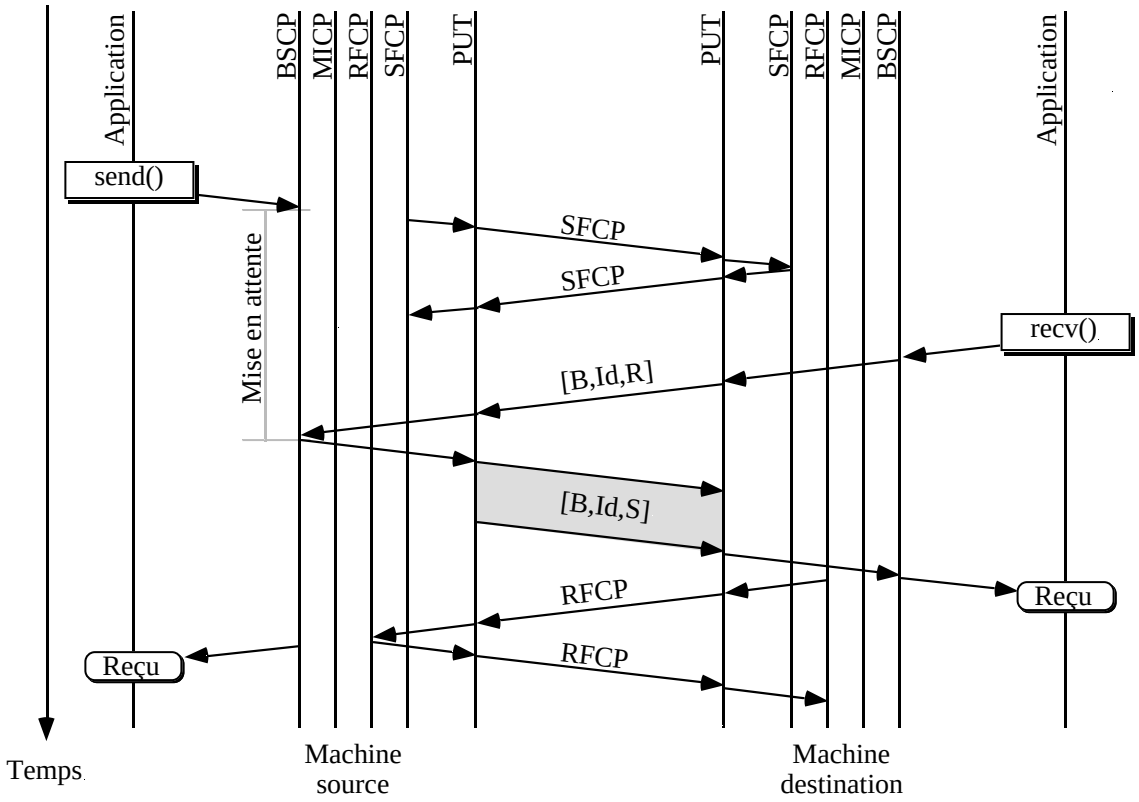

Fig. 6 : une transaction SCP/P sans faute. Comme il n'y a pas d'erreur, il n'y a pas de phase de récupération des MI, donc MICP n'est pas utilisé. On peut constater que chaque communication est gardée par un accusé, ce qui permet de détecter et de corriger les fautes.

IV-La machine MPC/2

Le but de cette partie est de présenter le nouveau conttrôleur réseau en cours de développement pour la machine MPC/2, ainsi que les perspectives offertes par ce nouveau contrôleur du point de vue de la sécurisation des communications.

IV.1-Un nouveau contrôleur réseau: ANI

Outre les problèmes de fiabilité du lien HSL, la machine MPC/1 souffre d'une autre limitation importante: le manque de programmabilité du matériel. En effet, pour régler un problème sur la machine MPC/1, on dispose de deux possibilités :

- Faire un nouveau circuit.
- Corriger le problème logiciellement.

Fabriquer un nouveau circuit est onéreux, et la perspective de devoir produire un nouveau PCIDDC à chaque fois que l'on fait un pas dans la phase de mise au point n'est pas réaliste. D'un autre côté, la solution de corriger tous les défauts matériels par le logiciel n'est pas idéale non plus. Les traitement logiciels additionnels prennent du temps et donc détériorent les performances, et nuisent considérablement à la lisibilité du code source. De plus, toutes les erreurs matérielles ne sont pas corrigeables logiciellement.

On a donc décidé d'utiliser un FPGA (*Field Programmable Gate Array*) comme contrôleur réseau, à la place de PCIDDC. Un FPGA est un circuit dont on peut programmer le comportement. Ce nouveau contrôleur réseau pour la machine MPC/2 a été nommé ANI (*Another Network Interface*).

Les bénéfices de la programmabilité d'ANI sont immédiats. Sur la machine MPC/2, il sera possible de corriger *matériellement* un problème matériel, et ceci à peu de frais. ANI permettra ainsi d'évaluer différentes pistes pour la résolution d'un problème donné, de réaliser des tests de performances, et de choisir la meilleur méthode.

De plus, passer à un nouveau contrôleur réseau nous permet d'éviter les erreurs commises avec PCIDDC. Par exemple, ANI s'interface avec la machine hôte par des accès mémoire et non plus par des accès en configuration, comme c'était le cas sur PCIDDC. Ceci permet d'éviter les dysfonctionnements obtenus lorsque l'on effectue des accès en configuration à une fréquence trop élevée sur une machine utilisant le jeu de composant Intel 440BX.

IV.2-Co-conception matérielle/logicielle

La présence dans la machine MPC/2 d'un contrôleur réseau reprogrammable tel que ANI ouvre de nouvelles perspectives du point de vue de la sécurisation des communications. Dans la machine MPC/1, on devait faire avec le comportement de PCIDDC, et inventer des mécanismes logiciels pour obtenir des communications sécurisées. Avec ANI, il est maintenant possible de changer le comportement du matériel pour que le logiciel trouve les services qui lui permettront d'assurer des communications sures de façon simple.

A chaque problème de sécurité, la question de la résolution matérielle ou logicielle du problème se pose. Faire un traitement en matériel a un avantage évident: la rapidité. De plus, tout le temps que le CPU ne passe pas à contourner des problèmes matériels, il peut le passer à calculer. On ne peut toutefois pas tout faire

matériellement, car la mémoire et les possibilités de traitement du FPGA ne sont pas aussi importantes que celles de la machine hôte. Des traitements trop complexes, ou manipulant des données trop importantes, doivent donc obligatoirement être faits logiciellement.

On est donc en face d'un travail de co-conception matérielle/logicielle. Il faut décider de ce qui doit être réalisé par le matériel et ce qui doit être réalisé par le logiciel.

IV.3-Sécurisation au niveau PUT ou SLR?

Le titre de cette partie a volontairement un air de déjà vu. En effet, puisqu'une nouvelle architecture est disponible, on peut se demander si l'on va à nouveau sécuriser au niveau SLR, ou si l'on va travailler au niveau PUT.

Il y a deux points à considérer. D'abord, la difficulté évoquée au III.2 : dans le cas de la sécurisation PUT, on devait gérer la possibilité de voir l'émetteur écraser un tampon déjà libéré sur le récepteur. Ceci était dû au fait qu'avec PCIDDC, nous n'avions aucune possibilité d'empêcher le dépôt d'une page sur le récepteur. Avec ANI, on a maintenant la possibilité de lever cette limitation, et de mettre en place un mécanisme de rejet des pages inattendues.

Cet argument plaidant contre la sécurisation au niveau PUT ne tenant plus, il reste à considérer la question des performances. SLR a un coût important en terme de performances. Lorsque l'on a porté PVM sur MPC [MPI], on s'est basé sur SLR. Le résultat a été suffisamment décevant en terme de performances pour que lors du portage de MPI sur MPC, on ait décidé d'utiliser PUT plutôt que SLR.

Ce deuxième point plaide en faveur d'une sécurisation au niveau PUT. Cela permettra d'obtenir des communications sécurisées, en gardant les meilleures performances possibles pour l'implémentation de MPI. Enfin, en sécurisant PUT, on sécurise automatiquement SLR, qui est bâti au dessus de PUT. Par contre, en sécurisant SLR, on ne sécurise pas PUT, et donc pas MPI, puisque MPI repose directement sur PUT.

V-Sécurisation au niveau PUT dans MPC/2

Dans cette partie, on propose les grandes lignes d'un mécanisme permettant la sécurisation de la primitive PUT pour la machine MPC/2. Les différentes options possibles sont d'abord examinées (type et niveau de la signalisation des erreurs sur le réseau), puis on étudie la réponse de la solution retenue à des événements tels que la perte d'un paquet ou la perte d'un acquittement. On aborde ensuite un passage en revue des différents cas particuliers posant un problème, ainsi que les possibilités de correction des dits problèmes. Enfin, cette partie se termine par l'étude de l'impact en termes de performances des solutions retenues, et une étude rapide de la sécurisation de SLR dans MPC/2

V.1-Détection des pertes de paquets

Il n'existe pas dans MPC/1 de moyen permettant au récepteur de détecter à coup sur la perte d'un paquet. Nous devons donc introduire un tel mécanisme.

Pour obtenir la détection des paquets perdus, il nous faut introduire des numéros de séquences : lorsqu'un nœud envoie des données, il place dans l'en-tête de chaque paquet un numéro de séquence, croissant unité par unité. De son côté, le nœud récepteur attend le bon numéro de séquence pour chaque paquet venant d'un autre nœud.

On a ainsi le moyen de détecter lorsqu'un paquet est perdu : le récepteur peut constater que la suite des numéros de séquences a été interrompue, et qu'un ou plusieurs numéros a été sauté. Ce mécanisme exige un comportement FIFO (*First In First Out*) du réseau, on doit donc renoncer aux tables de routage adaptatives.

V.2-Accusés de non-réception ou accusés de réception?

La sécurisation de PUT va obligatoirement passer par des échanges de paquets de contrôle entre machine source et machine destination, qui permettront à chaque machine de savoir où en est l'autre. Par ces paquets de contrôle, on va accuser réception ou non réception des données.

La première question que l'on peut se poser est le niveau de ces accusés: doit-on faire des accusés (de réception ou de non réception) des paquets, des pages, ou des messages? L'avantage de signaliser des éléments de petite taille tels que les paquets, est que l'on sait exactement ce qui est perdu. On peut donc en cas de perte ré-acheminer uniquement ce qui est nécessaire, ce qui économise de la bande passante. Par contre, la gestion de la chose se révélera nécessairement plus coûteuse en terme d'occupation mémoire, puisqu'il faudra garder des informations sur de nombreux paquets en transit.

Travailler au niveau message permet de réduire cette occupation mémoire. De plus, la perte d'un paquet étant un événement peu courant, il est sera rare d'avoir à ré-envoyer un paquet. L'économie en bande passante n'est donc pas très intéressante. On va donc préférer travailler au niveau message.

Il faut maintenant déterminer si l'on va utiliser des accusés de réception ou de non réception. La perte d'un paquet sur le réseau HSL étant un événement rare, l'utilisation d'accusé de non réception paraît un bon choix, car il va économiser le nombre de paquets dus à la signalisation. La figure 7 donne un exemple de transactions utilisant des accusés de non réception.

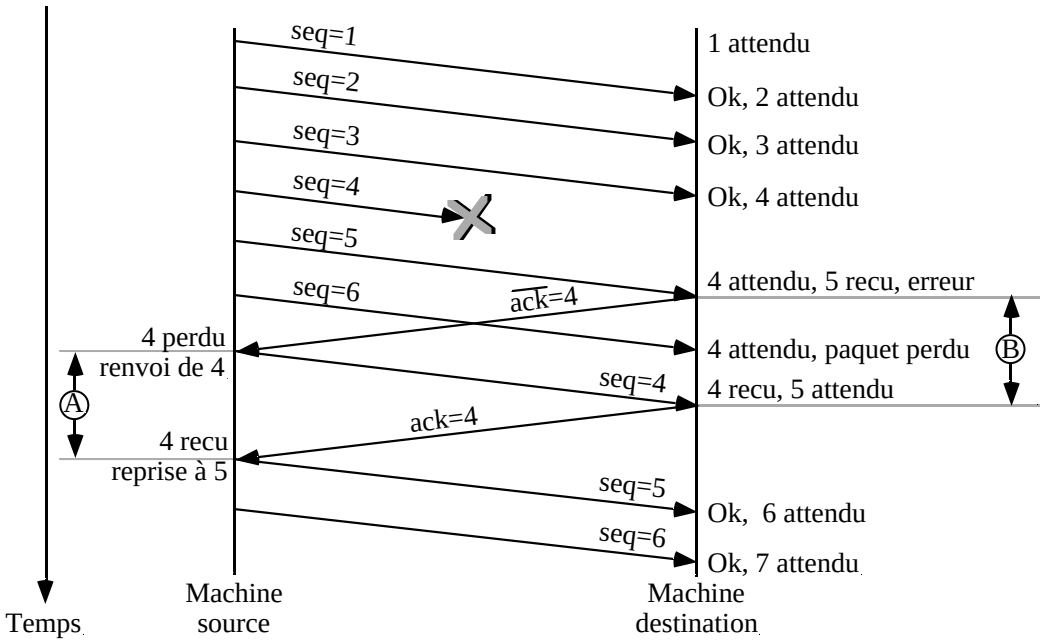

Fig. 7 : Perte de paquet et récupération sur erreur avec un accusé de non-réception. Chaque message est composé d'un seul paquet. On note une phase de récupération sur erreur sur l'émetteur (A) et sur le récepteur (B). Cette phase est terminée par un accusé de réception du paquet perdu. NB : le cas de la perte d'un accusé n'est pas abordé ici.

Plusieurs problèmes apparaissent dans cet exemple :

• Si le paquet perdu est le dernier d'un échange, aucun paquet portant un numéro de séquence inattendu ne sera reçu par le récepteur. Il ne demandera donc pas ré-expédition des informations perdues avant de recevoir un nouveau paquet. On peut dans certaines situations arriver à un interbloquage. La solution à ce problème serait d'avoir des paquets vides échangés à intervalle de temps régulier lorsque le lien est inutilisé, pour donner au récepteur l'opportunité de constater qu'un numéro de séquence a été sauté.

• Aussi bien sur l'émetteur que sur le récepteur, on a des phases de récupération sur erreur. Ce protocole, contrairement au PUT de MPC/1, ne laisse pas les nœuds sans états. Ceci implique une certaine complication dans l'implémentation du protocole

• Troisième problème, qui lui est insoluble: on signalise les non-réceptions, l'émetteur n'a donc aucun moyen de savoir quand un message a bien été *reçu* par le récepteur, et donc quand un tampon d'émission peut être libéré.

A cause du dernier problème, on ne peut pas utiliser d'accusés de non réception. Nous allons donc devoir élaborer une solution basée sur des accusés de réception. Si l'émetteur n'a pas reçu l'accusé de réception d'un message au bout d'un certain délais, il doit ré-envoyer le message.

V.3-Perte d'un paquet

La méthode de signalisation choisie est donc l'accusé de réception de chaque message reçu.

Du côté du récepteur, on observe les numéros de séquence des paquets reçus. Les paquets ayant un numéro attendu sont déposés en mémoire, les autres sont rejetés. Lorsque le dernier paquet d'un message est déposé en mémoire, le message est signalé comme reçu sur le récepteur, et un accusé de réception est envoyé à l'émetteur.

L'émetteur peut donc détecter la perte d'un paquet dans un message si il ne reçoit pas d'accusé de réception passé un certain temps. On va donc mettre en place un mécanisme de *timeout* : une fois un message envoyé, un compteur de temps est mis en marche. Si l'accusé de réception arrive avant le temps limite, le message est signalisé à l'utilisateur comme reçu. Sinon, le message est ré-envoyé, et le compteur de temps pour ce message est remis à zéro.

Pour que le mécanisme de numéros de séquence laisse passer le message envoyé pour la deuxième fois, il est indispensable de reprendre les numéros de séquences au numéro du premier paquet du message, et le découpage des pages du message en paquets doit être le même. On demande donc un découpage déterministe des messages.

La figure 8 présente deux transaction, l'une sans perte de paquet et l'autre avec.

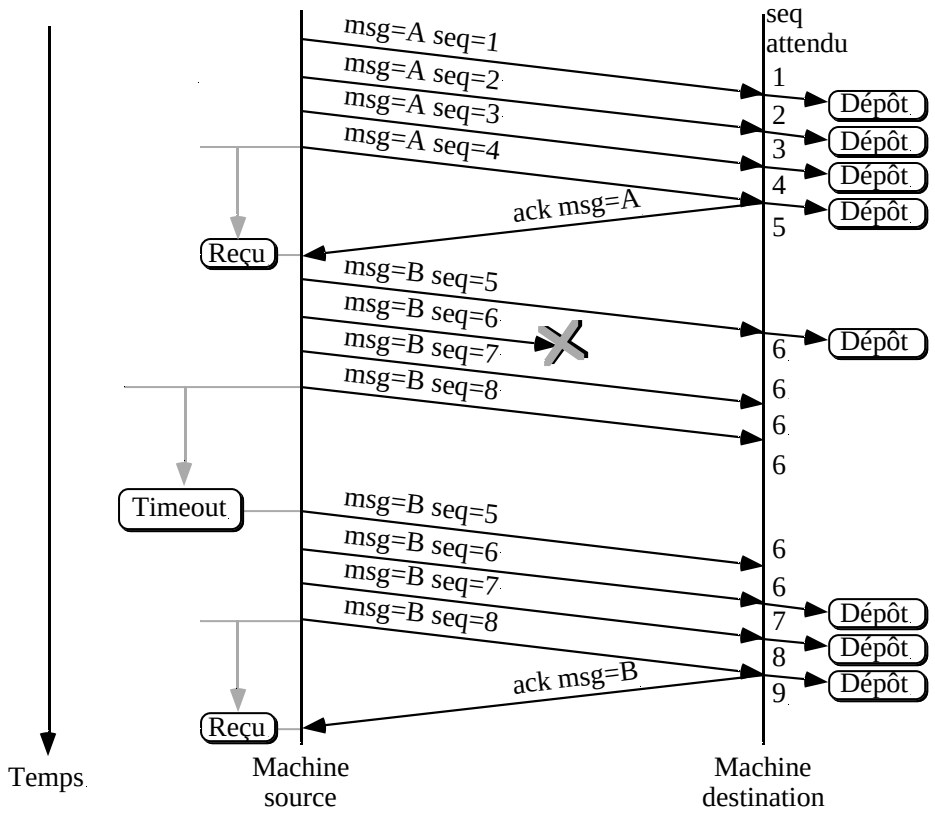

Fig. 8: Récupération sur la perte d'un paquet avec des accusés de réception. Pour chaque message envoyé, un compteur est mis en marche. Il est arrêté et le message est signalé "reçu" au retour de l'accusé de réception. Si au bout d'un certain temps l'accusé n'arrive pas ("timeout"*), le message est envoyé une deuxième fois. On note que la deuxième émission doit se faire avec les mêmes numéros de séquence que la première.*

On constate qu'avec cette méthode, on sait quand le message a été reçu, et donc quand on a la certitude de pouvoir libérer le tampon d'émission. Il faut aussi noter que le système des numéros de séquences

impose aux paquets de sortir du réseau dans l'ordre où ils y sont entrés. On requiert du réseau HSL qu'il se comporte en FIFO, c'est à dire qu'il ne soit pas adaptatif.

V.4-Perte ou retard d'un accusé de réception

Nous avons envisagé la perte d'un paquet, nous devons maintenant envisager la perte d'un accusé de réception. Lors d'une telle perte, le récepteur a déjà reçu le message, et il l'a signalisé à l'utilisateur. Le tampon en réception n'est donc pas utilisable. Si nous écrivons dans ce tampon, nous risquons de détruire des données écrites par l'application propriétaire du tampon. Par ailleurs, l'émetteur ne voyant pas arriver d'accusé de réception va finir par faire un *timeout* et renvoyer le message. Ce message doublon arrive au destinataire et ne doit pas être déposé dans la mémoire.

Nous avons un problème similaire si le réseau est particulièrement chargé et que le message et/ou son accusé de réception ont pris du retard dans la traversée du réseau. Si l'accusé de réception arrive après un *timeout*, l'émetteur ré-émet un un message qui a déjà été signalé comme reçue sur le récepteur.

Ce problème était difficilement soluble avec PCIDDC. Une contrainte de PCIDDC était que le récepteur ne pouvait empêcher le dépôt d'un paquet. Cette contrainte est levée avec ANI : nous pouvons savoir quels paquets nous avons reçu, et bloquer au niveau matériel le dépôt des paquets appartenant à un message doublon. Nous avons déjà le moyen de réaliser ce blocage : les paquets du message doublon n'auront pas le numéro de séquence attendu, et ils ne seront donc pas déposés. La figure 9 illustre ce propos.

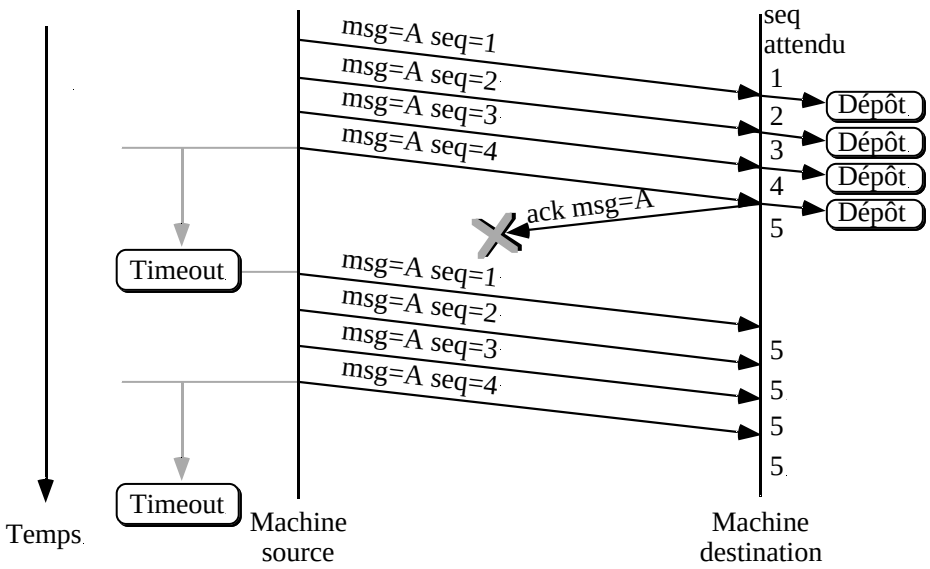

Fig. 9 : Perte d'un accusé de réception. Une fois le dernier paquet du message arrivé, le tampon en réception est libéré et l'accusé de réception envoyé. Si l'accusé est perdu, l'émetteur va re-émettre son message. Le tampon en réception ayant été libéré, on ne doit pas re-déposer ce message. Le système de numéros de séquence nous permet cela.

On a cependant un problème à corriger. Le deuxième envoi du message est rejeté, mais il doit tout de même donner lieu à un accusé de réception. Sans cela, le nœud émetteur ne recevra jamais l'acquittement du message, qui pourtant a été déposé et signalé sur le nœud récepteur. Il va donc répéter les tentatives d'envoi, et la réception du message ne sera jamais signalée sur le nœud émetteur.

Pour corriger cela, il faut introduire un drapeau indiquant que le message est une ré-émission (le drapeau bis). Lorsqu'il reçoit un message émis pour la deuxième fois, le récepteur sait que si le message est correctement reçu, il doit donner un accusé de réception même si le message n'a pas été déposé en mémoire.

La figure 10 donne un exemple d'utilisation du drapeau bis.

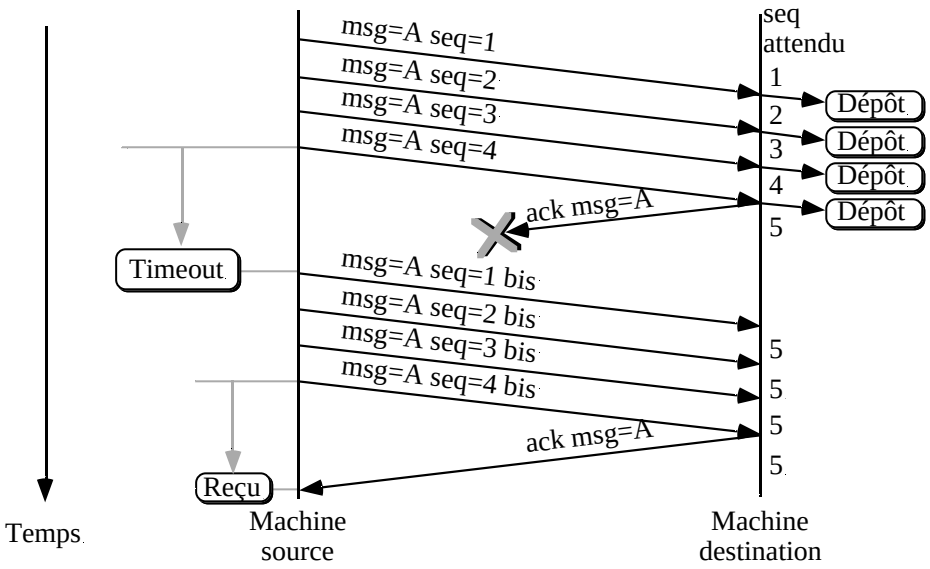

Fig. 10 : Utilisation du drapeau bis pour gérer une perte d'accusé de réception. Lorsque le drapeau bis est présent, le récepteur accuse réception du message, même s'il n'a pas été déposé en mémoire.

V.5-Condition d'envoi d'un accusé de réception.

Jusqu'au V.4, la condition d'envoi des accusés de réception était le dépôt en mémoire du dernier paquet d'un message. Maintenant, un accusé doit aussi être généré lorsqu'il n'y a pas de dépôt, et que le drapeau bis est présent. Examinons plus en détail la condition à laquelle le drapeau bis provoque l'envoi d'un accusé de réception. L'envoi de l'acquittement à la seule condition qu'un drapeau bis soit présent sur le dernier paquet d'un message conduit à une mauvaise signalisation. En effet, comme on peut le voir sur la figure 11, on acquitterait des messages non reçus intégralement.

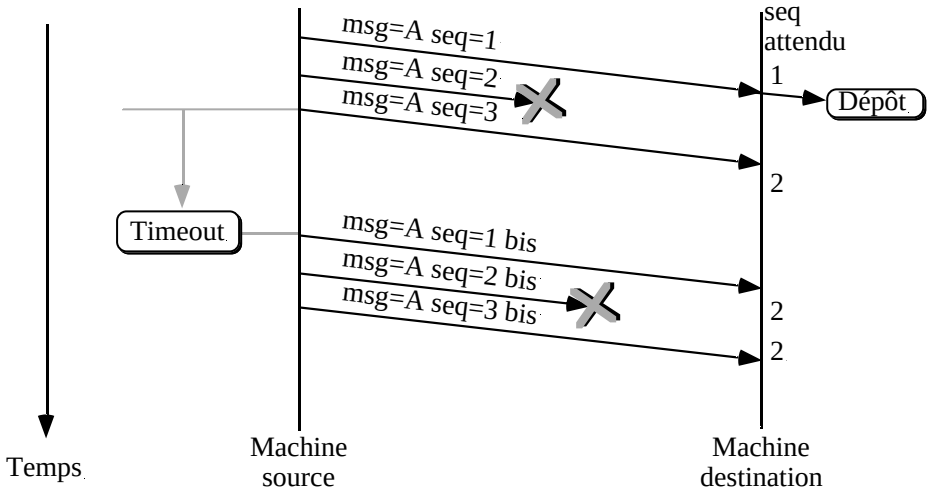

Fig. 11 : Un cas de double faute. Dans ce cas on voit bien qu'il ne suffit pas d'envoyer un acquittement lorsqu'on trouve le dernier paquet d'un message avec le drapeau bis.

En fait, la condition n'est même pas la réception intégrale du message avec le drapeau bis. Certaines conditions particulières font qu'on peut très bien recevoir en entier un message avec le drapeau bis et ne pas pouvoir envoyer d'accusé de réception. Une telle situation est décrite par la figure 12.

Le problème de la figure 12 peut être réglé si l'on empêche l'émetteur d'envoyer des messages entre une retransmission et son acquittement. Ainsi, le message B ne peut être retransmis avant que le message A soit acquitté, et on est ainsi sur qu'il sera accepté. Cette disposition nous donne l'échange décrit sur la figure 13.

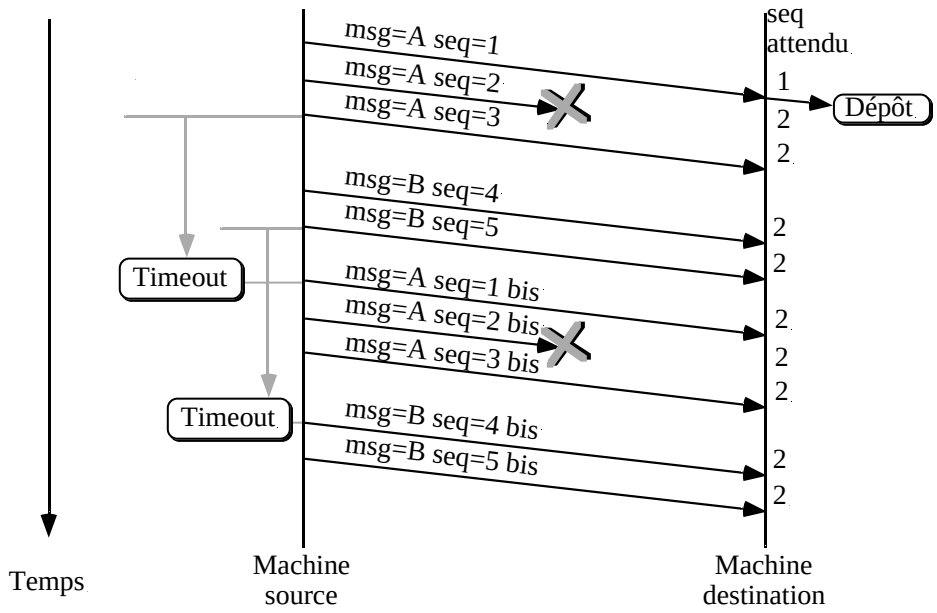

Fig. 12: Problème de double faute. On a une double perte du message A, avec le message B envoyé entre les deux transmission de A. B est reçu en entier et avec le drapeau bis, mais on ne doit pas l'acquitter pour autant, puisque on ne l'a pas déposé en mémoire.

Nous introduisons donc un état de reprise sur erreur sur l'émetteur. Pendant le mode d'opération normal, l'émetteur peut envoyer un ou plusieurs messages avant d'avoir obtenu l'accusé de réception du premier message. Lorsqu'il fait de la retransmission, les messages sont envoyés un par un, et on ne peut envoyer un message tant qu'on a pas eu accusé de réception du précédent. Dans ce mode, recevoir un message en entier signifie que le message a été déposé. En effet, la seule façon pour un message arrivé en entier de ne pas être déposé, c'est le cas où un paquet a été perdu dans un message précédent, et où le récepteur s'est bloqué. L'émetteur, qui n'a pas encore constaté le problème (il est en attente de l'accusé du message ayant subit la perte) peut continuer à envoyer des messages qui arrivent entier et qui ne sont pas déposés. Une fois qu'on est en mode récupération sur erreur, un message transmis en entier signifie un message déposé entièrement en mémoire.

Cet état de reprise sur erreur de L'émetteur a considérablement simplifié la situation. On peut ainsi donner une règle d'envoi d'accusé de réception qui conduira à une signalisation correcte : on doit accuser

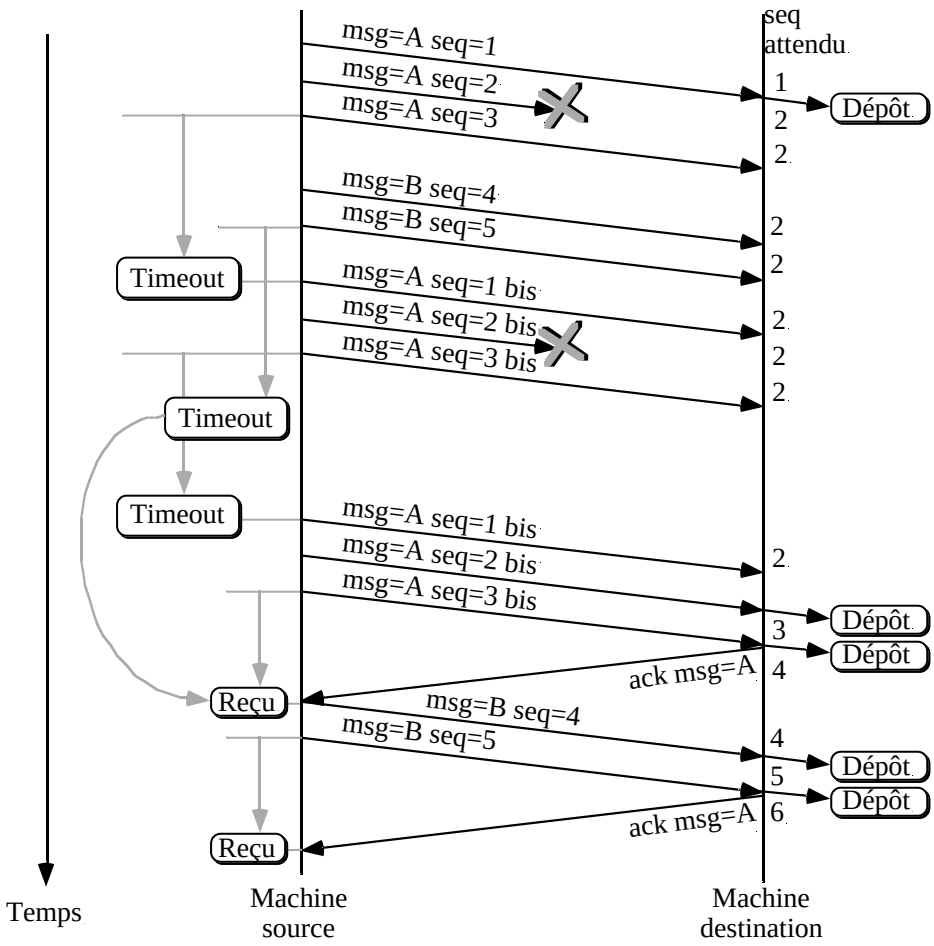

Fig. 13: Pas d'envoi de messages entre une retransmission et un acquittement. Ce procédé permet d'éviter le problème décrit à la figure 12..

réception d'un message lorsque l'une ou l'autre des deux conditions suivantes est remplie :
• Le dernier paquet du message a été déposé en mémoire
• Tous les paquets du message sont arrivés (en ayant été déposés ou non), tous avec le drapeau bis.

Le problème des conditions d'acquittement étant résolu, nous devons maintenant proposer un mécanisme qui permettra au récepteur de détecter que tous les paquets d'un message sont arrivés, tous avec le drapeau bis.

Détecter que tous les paquets d'un message arrivent revient à compter les paquets du message. Ce problème serait assez complexe s'il n'y avait pas l'atomicité de l'envoi des messages. Un nœud n'envoie qu'un message à la fois, et il n'entrelace jamais les paquets de deux messages différents. On est donc assuré que du point de vue du récepteur, on n'aura à compter les paquets que d'un seul message par nœud émetteur.

Si on rencontre le drapeau bis, même si les paquets ne sont pas déposés en mémoire, on va donc contrôler les numéros de séquences. Si on arrive jusqu'au dernier paquet du message sans jamais avoir sauté de numéro de séquence, alors tout le message a été envoyé avec le drapeau bis. La présence du drapeau bis indique que l'on est en mode de récupération sur erreur : l'émetteur n'envoie les messages qu'un par un, et il attend les acquittements avant de continuer. Comme on l'a expliqué plus haut, la conjonction de ce mode

de récupération sur erreur, et de la réception (éventuellement sans dépôt) de tous les paquets du message, implique que le message été entièrement déposé en mémoire, soit au cours des transferts pendant la phase de récupération sur erreur, soit avant, si la retransmission n'est due qu'à un accusé perdu, comme sur la figure 10. On peut donc acquitter le message si cette condition est remplie.

Dernier point à éclaircir : Il faut que le récepteur soit capable de détecter quel est le premier paquet d'un message. Sinon, il peut constater que les paquets d'un messages sont arrivés jusqu'au dernier, sans sauts dans les numéros de séquences, et conclure la réception de la totalité des paquets du message, alors que le ou les premiers paquets ont été perdus. On devra donc avoir un mécanisme permettant de savoir qu'un paquet est le premier d'un message. La mise en place de ce mécanisme sera examinée à l'implémentation, dans la partie VI.

V.6-Quelques cas pathologiques

Dans cette partie sont détaillées quelques transactions posant des problèmes, ainsi que les solutions proposées.

V.6.a-Une émission entre un acquittement perdu et la retransmission correspondante

Pendant une retransmission de message, on doit reprendre les numéros de séquences au numéro du premier paquet du message. Ceci a été illustré dans la figure 8. Toutefois, la figure 8 n'aborde pas le cas d'un autre message envoyé entre un acquittement perdu et la retransmission du message du à l'acquittement perdu. La figure 14 illustre ceci aussi clairement que possible.

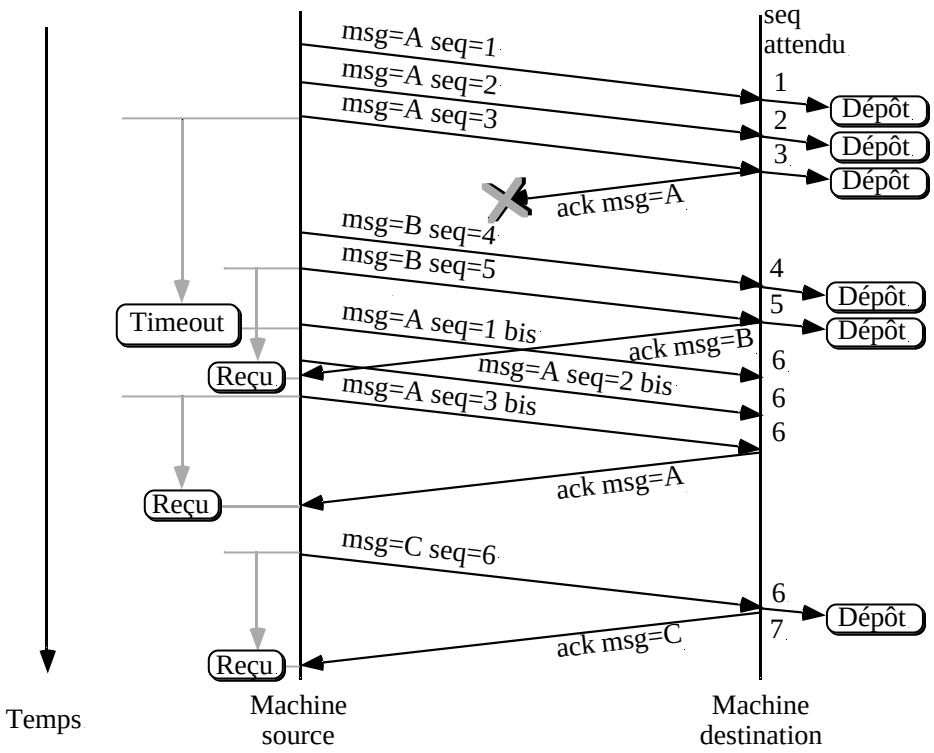

Fig. 14: Problèmes suite à la perte d'un accusé. Après la retransmission du message A, pour envoyer le message C, il faut reprendre les numéros de séquence où ils en étaient avant le retransmission, c'est à dire à 6, dans le cas présent, et non pas à 4.

On voit dans cette figure qu'après avoir fait la retransmission d'un message, il faut reprendre les séquences au numéro d'avant la retransmission.

V.6.b-Nœud destinataire inaccessible

Tout d'abord, il y a le cas de la machine inaccessible, par exemple parcequ'elle est éteinte ou parcequ'un câble est débranché. Dans ce cas, l'émetteur va ré-envoyer continuellement ses messages, sans jamais réussir à obtenir un accusé de réception, puisque le message n'arrive jamais à destination. Ce cas suggère fortement que le nombre de tentatives d'émission soit comptabilisé, et que l'émetteur abandonne l'émission au bout d'un certain nombre de tentatives infructueuses, signalant un échec à l'utilisateur.

Un problème peut survenir si on a trop hâtivement supposé que le nœud destinataire était inaccessible. On va alor recevoir un accusé de réception pour un message dont l'envoi a été abandonné. Il faut s'assurer que cet accusé ne pourra pas provoquer de problème. Dans le cas présent, on ne peut même pas vérifier l'absence d'accusé bloqué dans le réseau par un ping-pong (comme expliqué au III.3, si le réseau n'est pas adaptatif, un ping-pong permet de vérifier qu'aucun paquet n'est bloqué dans le réseau). En effet, le problème à détecter est l'impossibilité de contacter un nœud. Un ping-pong échouera donc comme l'envoi de n'importe quel autre message.

V.6.c-Congestion

Les problèmes de congestion imposent de bien choisir les délais avant retransmission (*timeout*). Si le délai est trop court, et que l'acquittement d'un message à été retardé ou perdu par un nœud congestionné, alors on va ré-envoyer un message doublon, alors qu'il a déjà été reçu par le récepteur. Ceci va augmenter la congestion. On a donc tout intérêt à utiliser un *timeout* élevé. On peut aussi songer à faire augmenter ce délai avec le nombre de tentatives de transmissions, pour essayer de diminuer la congestion.

V.6.d-Adaptativité

Nous avons expliqué plus haut la nécessité d'avoir un réseau non adaptatif. Mais cela n'est vrai que

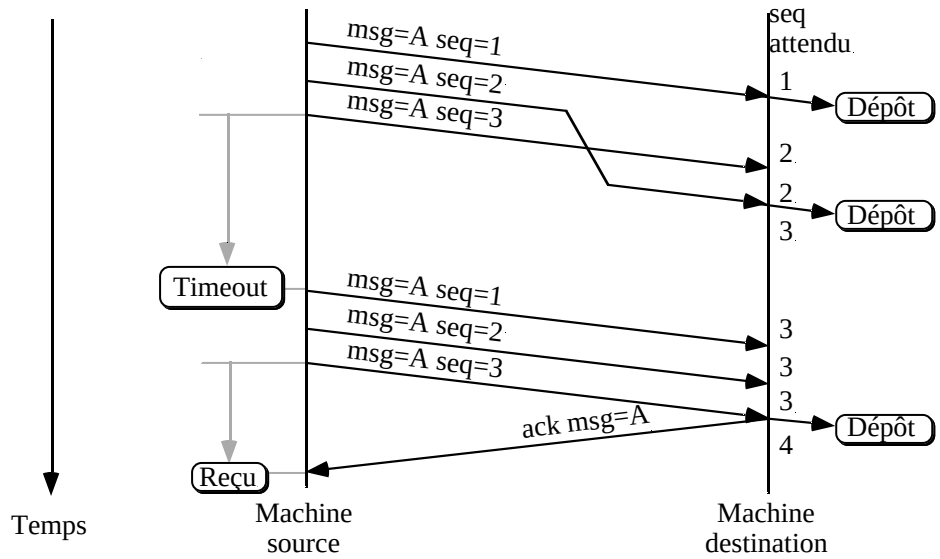

Fig. 15 : Réaction du protocole de sécurisation à un ré-ordonnancement des paquets. Le message arrive toujours à destination, mais au prix d'une retransmission.

dans une certaine mesure. En effet, un désordonnancement des paquets sera traité comme une erreur, et le système de numérotation des paquets va obliger les paquets à arriver dans l'ordre. On peut donc tolérer une faible adaptativité, la machine MPC ne fonctionnant que pendant les phases où le réseau se comporte de façon FIFO. La figure 15 donne un exemple de réaction à un ré-ordonnancement des paquets. Il est évident que moins le réseau se comportera comme une FIFO, plus on aura des erreurs et des retransmissions, et moins les performances seront bonnes. L'utilisation de table de routage adaptative paraît donc peu avantageuse, mais elle reste possible.

V.7-Impact sur les performances

Etudions l'impact des propositions précédentes sur les performances. Plusieurs aspects sont à prendre en compte: le travail du processeur pour gérer les communications, ainsi que la latence et le débit du réseau. étant donné que nous n'avons pas encore précisé ce qui doit être géré par le logiciel et ce qui doit être géré par le matériel, il n'est pas possible à ce stade de l'étude de préciser la charge de travail incombant au processeur. Intéressons-nous donc à la latence et au débit

V.7.a-Latence

La latence est augmentée par rapport au PUT de MPC/1. Dans MPC/1, le tampon d'émission était libéré dès que le dernier mot de 32 bits quittait PCIDDC. Maintenant, nous devons compter

- le temps d'écriture dans la mémoire distante des données en cours de transit dans le réseau.
- le temps de propagation de ce dernier mot jusqu'au contrôleur réseau de la machine distante
- le temps d'écrire ce dernier mot dans la mémoire distante.
- le temps que le processus transmetteur du nœud devienne disponible pour émettre un acquittement.
- le temps de traversée d'un accusé de réception jusqu'au contrôleur réseau de l'émetteur.

Au moment de l'envoi du dernier mot de 32 bits, il reste jusqu'à une centaine de ko de données en cours de transit sur le réseau (dans les files d'attentes des contrôleurs réseaux et des routeurs R-Cube). Mais pour calculer la latence, on s'intéresse au transit d'ue petit message de quelques octets. On n'a donc pas de données en cours de transit dans le réseau à prendre en compte.

Le temps de traversée d'un routeur R-Cube est évalué à 150ms. L'écriture d'un mot en mémoire centrale (en supposant que l'on avait déjà initié un transfert, ce qui est le cas ici) prend 30ns sur un bus PCI 33MHz/32bits. Enfin, le temps de traversée d'un contrôleur réseau est d'environ 100ms. [FW]

Et supposant que le réseau HSL soit fortement connexe, et donc qu'il n'y a que 2 routeurs R-Cube entre deux machines, nous obtenons donc un temps supplémentaire de latence par rapport à MPC/1 se décomposant ainsi :

Propagation du dernier mot:	
Traversée de deux R-Cube	2×150ns
Traversée des câbles	~10ns
Traversée de ANI	200ns
Ecriture en RAM	30ns
Envoi de l'accusé de réception	
Traversée de deux R-Cube	2×150ns
Traversée des câbles	~10ns
Total	~850ns

Nous devons donc compter une augmentation de latence de l'ordre de la microseconde dans l'utilisation de PUT. Le PUT de MPC/1 prenait environ 4µs, nous arrivons donc à 5µs, ce qui représente une augmentation de 25% de la latence. C'est une augmentation importante, mais c'est le prix de la sécurité: nous devons attendre la confirmation de la bonne réception d'un paquet avant de le signaliser comme envoyé chez l'émetteur et donc libérer le tampon d'émission. Ceci donne lieu à une augmentation de latence correspondant à un aller-retour sur le réseau.

Toutefois, l'augmentation de latence peut se révéler parfois plus importante. Dans le calcul précédent, nous n'avons pas compté le temps nécessaire au nœud récepteur pour avoir son processus d'émission disponible. Nous avons supposé que le récepteur n'était pas lui-même en train d'envoyer des données, et donc qu'il pouvait immédiatement envoyer l'accusé. Comptons maintenant ce qu'il se passe si le processus émetteur du nœud récepteur est en train d'envoyer des données.

On va choisir de donner aux accusés de réception une priorité maximum, et donc de permettre une émission des accusés de réceptions entre chaque émission de paquet. Si l'on limite la taille des paquets à 4ko (cette valeur correspond à une page pour le système de mémoire virtuelle), il nous faut au pire attendre le temps de transfert de 4ko pour pouvoir envoyer un accusé de réception. Dans la machine MPC/1, le goulot d'étranglement est le lien entre PCIDDC et R-Cube, mais il est probable que cette limitation sera levée dans MPC/2. On va donc supposer un goulot d'étranglement au niveau du bus PCI lui-même. Déterminons donc le temps nécessaire pour acheminer 4ko sur un bus PCI 33MHz/32bits.

On va supposer pour des raisons de simplicité que ce transfert sur le bus PCI se fait de façon idéale, c'est à dire que le bus est disponible quand on en a besoin. Un transfert sur le bus se fait en plusieurs opérations, qui suivent chacune les étapes suivantes:

Prendre le bus PCI	1 cycle
Phase d'adresse	1 cycle
Retournement du bus	1 cycle
Lecture de N mots de 32 bits	N cycles
Relâcher le bus	1 cycle
Total	4+N cycles

La durée du transfert (ici N cycles) est limitée par la valeur du registre PCI Latency Timer. Ce registre indique pendant combien de cycles un périphérique a le droit de garder le bus. Dans la machine MPC, pour maximiser les performances, on met ce registre à sa valeur maximum, c'est à dire 256 cycles. On va donc devoir faire 260 cycles pour ramener 256 mots de 32 bits, soit 1ko. Pour transmettre un paquet de 4ko, il nous faut donc 4 opérations comme celle là, soit, dans le cas idéal, 1040 cycles. Un cycle fait 30ns sur un bus à 33MHz, ce transfert prendra donc au pire 31,2µs.

On obtient donc dans ce cas une latence de 35µs au lieu de 4µs. C'est une augmentation énorme. De plus, cela n'est même pas une majoration : si le bus PCI n'est pas disponible dès qu'on en a besoin, la latence peut encore augmenter. En fait, le bus PCI ne garantit aucune majoration du temps nécessaire pour obtenir le bus, donc nous ne pouvons pas majorer la latence.

Cette valeur potentiellement très haute est due au fait qu'on ne peut envoyer deux paquets en même temps de ANI à R-Cube. La seule façon de la faire diminuer serait de diminuer la taille des paquets. En prenant des paquets de 256 octets, par exemple, la latence maximum, dans le cas idéal où le bus PCI est disponible tout de suite, passe à 12µs. On peut avoir l'idée de faire des paquets très petits, de quelques dizaines d'octets, mais diminuer autant la taille des paquets ne serait pas raisonnable, car le temps passé à initier les opérations sur le bus PCI pèserait de plus en plus lourd par rapport au temps passé à transférer des données. Avec une taille de paquet de 256 octets ou plus, on maximise le temps passé à transférer des données.

V.7.b-Débit

En terme de débit, l'envoi d'un accusé de réception pour chaque message occupe de la bande passante. Lorsqu'il s'agit d'une topologie fortement connexe, il y a forcement un lien non partagé entre émetteur et récepteur. De deux choses l'une:

• Soit le lien retour n'est pas chargé: dans ce cas, on peut envoyer des accusés de réception dessus sans diminuer les performances

• Soit le lien retour est chargé: on pourrait alors placer les accusés de réception dans les en-tête des paquets passant sur le lien retour. On ne produirait ainsi pas de charge supplémentaire due aux accusés de réceptions.

Un problème survient toutefois lorsque le réseau n'est pas fortement connexe. On peut arriver à des trafics tels que celui décrit par la figure 16.

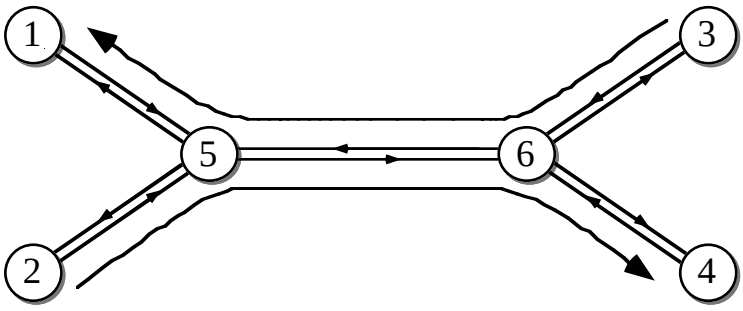

Fig. 16 : exemple de topologie de réseau HSL non fortement connexe. On a deux flux de données importants, l'un de 3 vers 1, et l'autre de 2 vers 4. Le lien entre 5 et 6 est partagé pour ces deux flux.

Avec un tel traffic, on n'a pas de données allant de 4 vers 2 pour placer les accusés de réception des messages allant de 2 vers 4. On devra donc envoyer des paquets vides ne contenant que des accusés de réception. Mais ces paquets vont occuper de la bande passante sur le lien de 6 vers 5, qui est déjà chargé par les transferts de 3 vers 1.

Evaluons la bande passante occupée en sens inverse par les accusés de réception. Elle est directement liés au débit en message par seconde, puisqu'un accusé est envoyé pour chaque message reçu. Calculons le débit maximum en message par seconde, en fonction de la taille des messages. Le goulot d'étranglement sur le chemin de données étant le bus PCI, la vitesse d'envoi des messages est limitée par la vitesse de lecture

des entrées de LPE et des données du message. Avec un message de n octets (n inférieur à 1ko), on obtient ceci:

Lecture de l'entrée de LPE	
Prendre le bus PCI	1 cycle
Phase d'adresse	1 cycle
Retournement du bus	1 cycle
Lecture de l'entrée (4×32 bits)	4 cycles
Lâcher le bus	1 cycle
Lecture des données à emmetre	
Prendre le bus PCI	1 cycle
Phase d'adresse	1 cycle
Retournement du bus	1 cycle
Lecture des données (E(n/4)×32 bits)	E(n/4) cycles
Lâcher le bus	1 cycle
Total en cycles	(12+E(n/4))

Avec un cycle PCI à 30ns, on obtient un débit maximum de 1 message court envoyé toutes les 30×(12+E(n/4)) nanosecondes, et donc un accusé de réception renvoyé tout les 30×(12+E(n/4)) nanosecondes. Supposons un paquet de 12 octets pour les accusés de réception (nous verrons plus loin les raisons qui conduisent a choisir cette taille). La bande passante occupée en sens inverse est alors donnée par la formule suivante :

$$D_{Mb/s} = \frac{8 \cdot d_{ack}}{1024^2 \cdot T_{PCI} \cdot \left[12 + E\left(\frac{n}{4}\right)\right]}$$

d_{ack} = 56 octets, taille des acquittements
T_{PCI} = 30ns, durée d'un cycle PCI

Ce débit, pour des petits messages, est colossal: environ 1,2Gb/s. Il faut cependant tempérer ce résultat: il s'agit d'une majoration. Cette formule suppose une situation idéale, où le bus PCI est toujours disponible, et où le logiciel est capable d'ajouter des entrées de LPE aussi vite que le matériel les consomme.

D'après les différents tests réalisés sur la machine MPC/1 [ED], on sait bien que cette dernière supposition est erronée. Sur la machine MPC/1, le logiciel est incapable de suivre la vitesse du matériel pour des petits messages. La taille limite est de l'ordre du millier d'octet. La vitesse d'émission des petits messages est donc majorée par la vitesse d'émission des messages de 1ko. Le débit utilisé par les accusés de réception est donc majoré par 53Mb/s, d'après la formule établie ci dessus.

Ce chiffre, s'il est plus raisonnable que les 1,2Gb/s initiaux, reste élevé. Il ne faut pas oublier qu'il s'agit d'un majorant. Non seulement nous supposons le bus PCI toujours disponible, mais en plus, il correspond à une utilisation assez artificielle de la machine, où on essaye d'émettre le plus de données possible le plus rapidement possible. De tels comportements se rencontrent rarement dans des applications normales de calcul parallèle, on ne les trouve en réalité que dans les programmes de tests de performances de notre machi-

ne.

La forte bande passante potentiellement occupée par des accusés de réception suggère de faire une légère amélioration du système d'accusés : il s'agirait, lorsque cela est possible, d'accuser réception de plusieurs messages à la fois. Des paquets d'accusés de réceptions multiples devraient donc être envoyés lorsque le débit utilisé pour accuser réception des messages devient trop important. Cette possibilité n'est pas approfondie dans le présent document, mais elle constitue une piste possible pour optimiser les ressources utilisées par la sécurisation des communications.

Il faut toutefois noter que les suggestions d'optimisation sur les accusés de réception ne sont pas forcément intéressantes. Si le mécanisme d'acquittement est géré logiciellement, alors effectivement, l'envoi des acquittements dans les en-têtes des paquets allant en sens inverse, ainsi que l'agrégation des acquittements, seront des idées à retenir. Par contre, si on doit gérer ce mécanisme matériellement, ces optimisations deviennent extrêmement coûteuse. Ce qui se fait facilement dans la matériel, ce sont les opérations orthogonales : les collaborations entre tâches matérielles sont toujours un problème délicat. On n'essayera donc pas de les mettre en place si on arrive à une solution où les acquittements sont gérés par le matériel.

V.8-sécurisation au niveau SLR

Examinons ce qui se passerait si on tentait de sécuriser la machine MPC/2 au niveau SLR. La figure 4 décrit un transfert dans les deux cas possibles: *send()* avant le *receive()*, et *receive()* avant le *send()* .

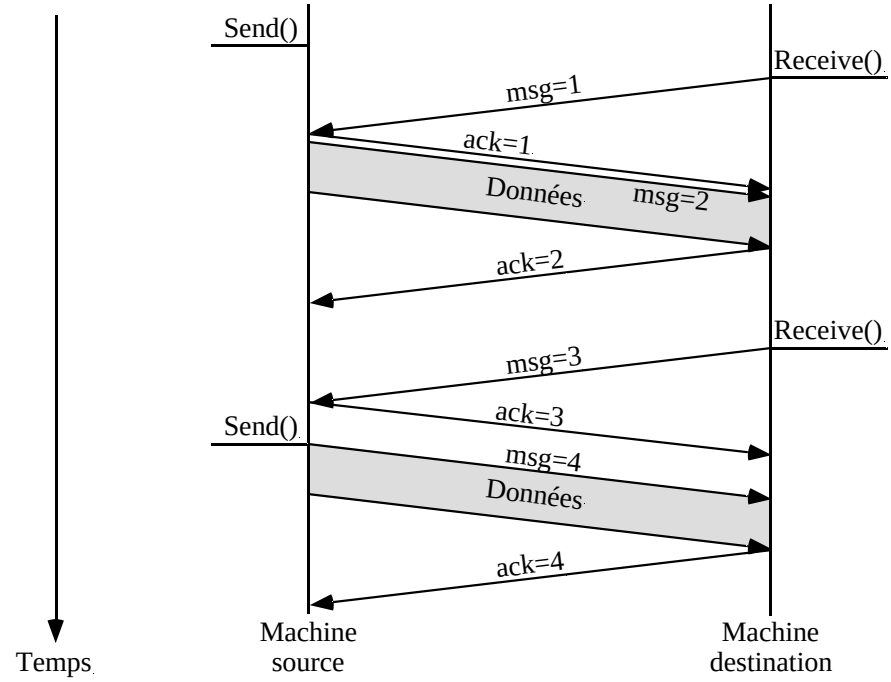

Fig. 17 : Deux échanges SLR basés sur PUT sécurisé.

Maintenant, examinons sur la figure 17 ce que donne SLR au dessus du PUT sécurisé proposé dans le présent document. Si on essaye de faire un SLR sécurisé sur PUT non sécurisé, c'est parce qu'on espère qu'avec une vision plus haut niveau des échanges, on pourra faire des optimisations dues à des situations par-

ticulières. Le but est donc de prendre les échanges SLR sur PUT sécurisé, et essayer de supprimer des messages. On constate rapidement qu'on ne peut pas faire beaucoup d'économies.

Dans le cas où le *send()* est avant le *receive()*, les ajouts au SLR sur PUT traditionnel sont les accusés de réception des données (ack=1 et ack=2 sur la figure). Le ack=2 est indispensable, pour permettre à l'émetteur de savoir quand il peut libérer le tampon d'émission et signaliser la complétion du *send()*. Avec une vision de plus haut niveau, on pourrait agréger le ack=1 dans l'envoi des données (msg=2 sur la figure)

Dans le cas où le *send()* est après le *receive()*, on a toujours l'accusé de réception des données (ack=4 sur la figure), que l'on ne peut pas supprimer pour la raison exposée précédemment pour ack=2. Reste l'accusé de réception du message de contrôle (ack=3 sur la figure), que l'on pourrait espérer supprimer, en l'agrégeant avec le message de données (msg=4 sur la figure). Ceci n'est hélas pas possible, car rien n'impose que le *receive()* et le *send()* soient rapprochés dans le temps. L'accusé (ack=3) est donc indispensable, car il assure la sécurité du message de contrôle (msg=3), même si le *receive()* survient plusieurs minutes avant le *send()*.

Sécuriser au niveau SLR plutôt qu'au niveau PUT ne permet donc l'économie que d'un accusé de réception. Ceci ne permet pas de gagner en latence, mais seulement en débit. Il s'en suit que la sécurisation au niveau SLR plutôt que PUT n'est pas très intéressante, *du simple point de vue de la sécurité*. On ne gagnerait pas en performances à avoir un SLR sécurisé et implémenté dans le matériel sur un PUT non sécurisé, plutôt qu'un SLR matériel sur un PUT matériel sécurisé. Par contre, il est certain qu'une implémentation des couches SLR gérée par le matériel plutôt que par le logiciel permettrait de gagner beaucoup en performances au niveau SLR. Reste à savoir si SLR est implémentable dans le matériel. Cette question sort du cadre de la présente étude.

VI-Proposition d'implémentation de S/PUT-1.0

Dans cette partie est détaillée une proposition d'implémentation pour la primitive S/PUT (Secured PUT). Ne doutant pas que cette spécification aura ses limites, et que des modifications devront y être faite, elle porte déjà un numéro de version : 1.0. Au cours de cette partie, on reprends les besoins proposés dans la partie précédente, et on étudie la façon dont on pourrait les implémenter, logiciellement ou matériellement. Son abordés le mécanisme de renvoi des messages non acquittés, l'utilisation des numéros de séquences sur les paquets, ainsi que les condition et les mécanismes d'envoi des accusés de réception. Enfin, le VI-6 propose une sythèse de S/PUT, résumant de façon succinte les mécanismes et comportement mis en place jusque là.

VI.1-Implémentation des renvois: la table des messages en transit

Nous avons dans la partie précédente identifié un besoin sur l'émetteur: celui de re-envoyer les messages non acquittés par le récepteur après un certain temps. Avant de nous intéresser à la répartition des tâches entre logiciel et matériel, précisons plus avant la façon dont un tel mécanisme peut être mis en place.

Ce dont nous avons besoin, c'est un compteur de temps pour chaque message envoyé. Un accusé de réception arrête le compteur, et si le compteur atteint une valeur maximum, il faut re-envoyer le message. Evaluons le nombre de compteurs dont nous pouvons avoir besoin simultanément.

Pour cela, nous devons dimensionner la durée du *timeout*. Cette durée dépends du temps que met un accusé de réception à arriver, après que l'émetteur ait envoyé le dernier octet d'un paquet. Nous avons évalué ce temps à environ 850ns dans le V.7.a. Une première idée serait de régler le délai de ré-émission à 850ns. Mais ce serait une grave erreur : tout d'abord, ce chiffre de 850ns est un chiffre idéal, correspondant à une situation où les bus PCI sont toujours disponibles sans délai, où le récepteur peut envoyer un accusé de réception immédiatement, et où il n'y a aucun retard dû à de la congestion dans le réseau HSL. Nous devons donc utiliser une valeur supérieure à 850ns.

De plus, la perte d'un paquet étant un événement rare, avoir un *timeout* sur-dimensionné n'a pas un impact énorme sur les performances. Par contre, un *timeout* trop petit provoquera de multiples re-émissions de paquets dus à des retards d'accusés de réception. Si ces retards sont liés à de la congestion sur le réseau HSL, on va saturer encore plus le réseau. Avoir un *timeout* court est donc définitivement une idée à rejeter.

La question reste : comment dimensionner le délai de re-émission? On a vu qu'on pouvait avoir à attendre 31µs pour que le récepteur soit en mesure d'envoyer un acquittement. Si on s'intéresse aux problèmes d'obtention du bus PCI, on constate qu'on peut attendre jusqu'à 255 transactions de 30ns par périphérique présent sur le bus avant d'obtenir le bus. Ceci nous fait 76,5µs au pire, si on compte 10 périphériques sur le bus. Ensuite, nous avons la traversée du réseau HSL. Là, le problème est pire encore, un paquet pouvant rester bloqué un temps infini dans le réseau (si un lien est brisé, par exemple). En fonctionnement normal, si les paquets sont assez gros, on peut compter 2ko de données en cours de transit dans le réseau à acheminer entre le départ du dernier octet de l'émetteur et l'arrivée du dernier octet sur le récepteur. Ces 2ko de données donnent lieu à 15µs de temps d'attente supplémentaire. Nous devons donc prendre une valeur arbitraire, en espérant qu'elle sera un bon majorant. Prenons un *timeout* de $\Delta = 500\mu s$, ce qui fait 1000 fois le temps correspondant au cas idéal.

Une étude statistique serait la bienvenue pour tester la validité de cette valeur. Pour réaliser cette étude, on a besoin de données expérimentales telles que l'occupation du bus PCI pendant l'utilisation normale de la machine MPC, ou le temps de traversée d'un paquet à travers le réseau HSL. Une autre approche, plus pragmatique, consisterait à essayer différentes valeurs jusqu'à l'obtention de résultats satisfaisants. Etant donné que le FPGA est reprogrammable, cette dernière solution sera sans doute retenue.

Calculons le nombre de messages que nous pouvons envoyer pendant le temps Δ, et donc de le nombre de compteurs que nous devons maintenir. Pour ce calcul, on suppose le cas idéal, car on veut un majorant du nombre de messages que l'on peut envoyer en un temps Δ. On a vu que la vitesse d'émission des messages était majorée par la vitesse d'émission des messages de 1ko, soit un message tous les 8µs. En 500µs, 62 messages peuvent être envoyés. Ceci implique que l'on doit être capable de faire tourner 62 compteurs à la fois.

Que les compteurs soient gérés en logiciel ou en matériel, il n'est pas question de faire fonctionner tant de compteurs à la fois, cela occuperait trop de ressources. Au lieu de ça, on va exploiter le fait que tous ces compteurs ont un point commun : c'est le temps qui les fait avancer, tous au même rythme. On va donc pouvoir les remplacer par un compteur unique. Deux solutions d'implémentations sont disponibles : une table indexée par le temps, ou une table contenant des champs d'échéance.

VI.1.a-Table des messages en transit indexée par le temps

Lorsque l'émetteur traite une entrée de LPE à l'instant t, il va placer les informations relatives à cette entrée à l'index t dans la table des messages en transit. Voici les champs composant une entrée:

- Le numéro de l'entrée de LPE traitée
- Le numéro de LPE (si on a des LPE multiples)
- Le numéro de séquence du premier paquet du message
- Le nombre de tentatives d'envoi de l'entrée

Par ailleurs, l'émetteur maintient un compteur, qui est incrémenté toutes les 240ns (c'est le temps minimum nécessaire pour lire une entrée) lorsqu'il n'est pas en train d'émettre un message, ou entre chaque émission de message, si il est en train d'émettre. Ce compteur est synchronisé sur les cycles du bus PCI. Appelons ce compteur compteur de temps.

Chaque message envoyé porte dans ses en-têtes son numéro d'index (le msg dans les figures de la partie V). C'est l'index du message dans la table des messages en transit. Il faut noter que ce numéro d'index n'a rien à voir avec l'identifiant de message (MI) utilisé dans l'API PUT de MPC/1. Contrairement aux MI, ce numéro d'index est totalement géré par le système, et l'utilisateur n'y a pas accès. Le mécanisme des MI est parallèlement maintenu, et il n'est utilisé nulle part par S/PUT. Conformément aux observations concernant le champ MI dans la partie II.7.c, nous le laissons entièrement à l'utilisateur.

Lorsque le récepteur reçoit un message, il renvoie un accusé de réception. Dans cet accusé, il y place l'index du message. L'émetteur du message recevant l'accusé de réception, prend connaissance du numéro d'index IDX du message qui est acquitté. Il peut alors aller lire dans la table des messages en transit, à l'index IDX, l'index du descripteur de LPE de la première page du message. L'entrée correspondante dans la table des messages en transit est libérée, et l'émission du message est signalée à l'utilisateur.

Si aucun accusé de réception n'arrive, l'entrée correspondant au message reste dans la table des mes-

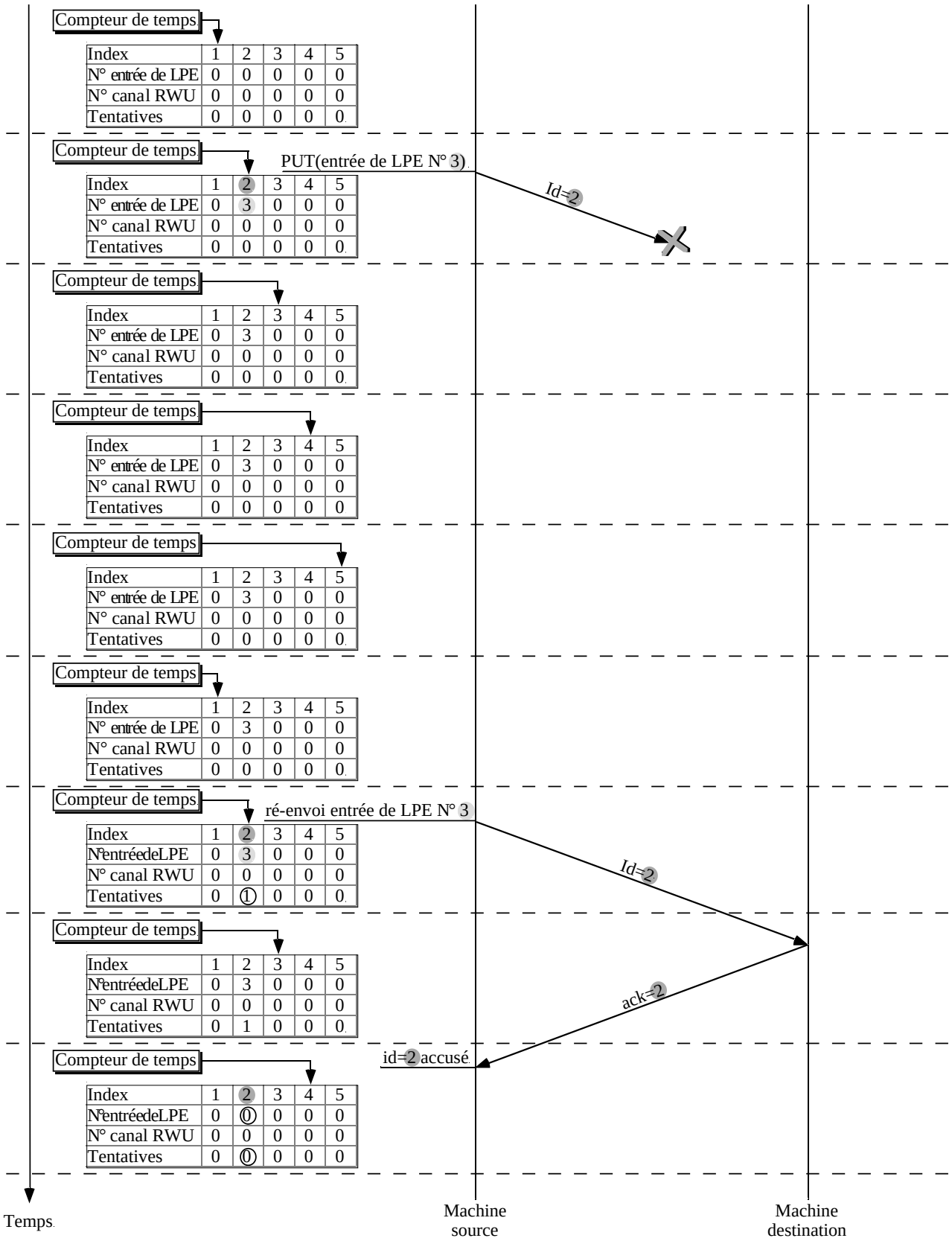

Fig. 18 : Table des messages en transit indexée par le temps, lors d'un envoi avec erreur. Par souci de clarté, on s'est limité à une table du Temps à 5 entrées, les durées des temps d'émission et d'accusé de réception ne correspondent pas aux valeurs réelles, et les messages font tous un paquet.

sages en transit. Au bout d'un moment, le compteur de temps arrive à la dernière entrée de la table. Il est alors re-initialisé, et le travail continue en reprenant à la première entrée de la table. Arrive alors un moment où le compteur de temps va pointer sur l'entrée correspondant au message qui n'a pas été acquitté. C'est le moment de la re-émission du message.

A chaque retransmission de message, le compteur du nombre de tentatives d'envoi de l'entrée doit être incrémenté. Ceci permet l'arret des tentatives d'envoi du message au bout d'un certain nombre de tentatives, et de signaler à l'utilisateur un problème important (probablement une machine inaccessible).

La figure 18 illustre l'évolution de la table des messages en transit lors d'un échange avec perte d'un message. Le cas de la perte de l'accusé de réception est similaire au cas de la perte d'un paquet.

Les traitements à effectuer sur la table des messages en transit sont extrêmement simples. On peut les implémenter matériellement, à condition que la table elle-même ne soit pas de taille trop importante.

Cette méthode présente l'avantage qu'un message retransmis a le même index dans la table que le message original. Si un accusé de réception arrive très en retard, après la re-émission du message, c'est tout de même le bon message qui sera acquitté. Deux inconvénients sont à noter:
• La taille de la table doit être adaptée à la durée du *timeout*. Ceci pose un gros problème car on ne peut pas jouer sur la durée du *timeout* comme on le voudrait sans agrandir considérablement la table.
• Pour ne pas sauter des entrées, on doit arrêter l'avancée dans la table pendant l'emission d'un message (parce qu'on ne peut pas ré-émettre un message pendant l'émission d'un autre message). L'index dans la table n'est donc ni le temps, ni le numéro du message envoyé, c'est une quantité hybride.

Pour résoudre ces deux inconvénients, on propose l'utilisation d'une table qui n'est pas indexée par le temps, mais qui contient des champs d'échéances.

VI.1.b-Table des messages en transit avec champ d'échéance

La méthode proposée ici est dérivée de la méthode utilisée dans 4.4BSD pour gérer les *timeouts* dans le système [BSD]. Il s'agit d'utiliser une liste chaînée. Chaque élément de la liste contient une date d'échéance en temps relatif, une action, et bien-sûr des pointeurs vers les éléments suivants et précédents de la liste.

Le champ d'échéance indique combien de temps après l'entrée précédente l'entrée courante doit être traitée. Le système possède un pointeur sur la première entrée, et il sait en regardant le champ d'échéance de la première entrée dans combien de temps il doit exécuter son action. Une fois la première entrée exécutée, elle est supprimée de la liste, et on recommence de même avec la seconde entrée. La figure 19 indique comment marche la liste chaînée.

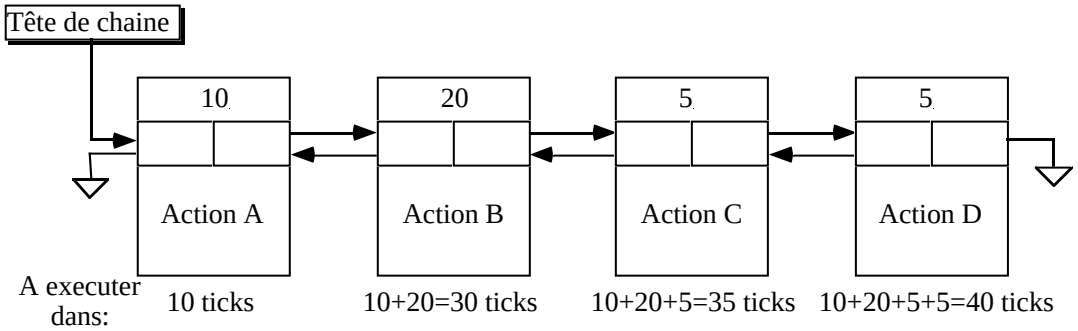

Fig. 19 : les listes chaînées gérant les timeouts *dans 4.4BSD. Le tick est l'unité de temps.*

Dans l'exemple de la figure 19, pour ajouter une action, par exemple à exécuter dans 32 ticks, il suffit d'insérer un élément dans la chaîne, entre le deuxième et le troisième, avec une échéance de 2, et de modifier l'échéance de l'élément suivant pour la faire passer de 5 à 3. De même on peut supprimer un élément, moyennant d'augmenter le champ d'échéance de l'élément suivant. Lorsque l'on ajoute ou on supprime des éléments dans la chaîne, la somme des champs d'échéance des éléments doit rester constante, sauf si l'opération porte sur le dernier élément ou si il s'agit de l'exécution du premier élément.

Ce mécanisme, tel qu'il est employé dans 4.4BSD est très général. On peut l'utiliser pour gérer n'importe quelle situation. Pour ce qui nous concerne, une faible partie de ses fonctionnalités nous intéresse, car on est dans un cas particulier : le *timeout* etant fixe, les nouveaux éléments sont toujours ajoutés à la fin de la chaîne, jamais au milieu. On peut donc se passer de l'aspect liste chaînée, et se contenter d'utiliser une table. Ceci a son importance, car on désire si possible gérer ce mécanisme matériellement.

La table des messages en transit va donc s'organiser comme suit : chaque entrée contient les champs suivants:
• Echéance (T)
• Nombre de tentatives d'envoi (R)
• Le numéro de séquence du premier paquet du message
• Le numéro de l'entrée de LPE traitée
• Le numéro de LPE (si on a des LPE multiples)

Ajoutons à cela un certain nombre de quantités permettant de gérer la table:
• Le temps courant (noté t)
• Le délai de retransmission (*timeout*, Δ)
• Le nombre d'entrées dans la table (n)
• Le pointeur de première entrée libre (N)
• Le pointeur du plus ancien message (F)
• Le pointeur sur le prochain message en *timeout* (W)

Enfin, notons des conditions importantes:
• La table est vide : $N = F$
• La table est non pleine : $N+1 \neq F$ [n]
• Un élément est valide (il décrit un message en transit) si $R \neq 0$, sinon il est invalide.
• Un élément décrit un message en instance de retransmission s'il a $T = 0$ et $R \neq 0$.
• Les éléments valides sont entre F (inclu) et N (exclu).
• Les éléments en instance de retransmission sont entre F (inclu) et W (exclu).
• Les éléments valides en attente d'acquittement sont entre W (inclu) et N (exclu).

La table est manipulée par trois processus, qui ont au total 4 comportements :.

• Le processus d'émission, en mode normal, ajoute une entrée dans la table lorsqu'il a envoyé un message. Pour cela, il initialise l'entrée T à Δ, et l'entrée R à 1 (première tentative).

• Le processus de surveillance (*Watchdog*) observe l'entrée pointée par W et incrémente t tous les ticks (1tick=240ns). Lorsque t devient égal au champ T de l'entrée pointée par W, il indique que le message décrit par l'entrée est à renvoyer (en mettant le champ T à 0), ré-initialise t à 0, et passe à l'entrée suivante en incré-

mentant W. Si W devient égal à N, il n'y a plus de message en transit à surveiller, et *watchdog* s'endort en attendant l'envoi d'un nouveau message.

• Le processus de réception, lorsqu'il reçoit un accusé de réception, supprime l'entrée correspondante, en mettant son champ R à 0, et en reportant l'échéance sur l'entrée valide suivante. Si c'est l'entrée pointée par F qui a été acquittée, F doit être incrémenté pour pointer sur l'entrée valide suivante.

• Lorsque le processus de surveillance marque une entrée comme étant en instance de retransmission, un compteur, nommé $\sum R$, est incrémenté. Lorsque ce compteur est non nul, le processus d'émission sait qu'il doit en priorité renvoyer les messages en *timeout* avant de pouvoir envoyer de nouveaux messages. Il va donc parcourir la table à partir de l'entrée pointée par F et jusqu'à l'entrée pointée par W, à la recherche de messages à renvoyer. A chaque message renvoyé, l'ancienne entrée est invalidée, et une nouvelle entrée est créee à l'endroit pointé par N. De plus, $\sum R$ est decrémenté, de telle façon que le processus d'émission sait quand il peut retourner à l'émission des entrées normales (lorsque $\sum R=0$). Pour les raisons développées au V.5, pendant les retransmissions, le processus émetteur attend l'accusé de réception de chaque message avant de transmettre le suivant.

La figure 20 donne un exemple d'utilisation de cette deuxième version de la table des messages en transit, avec les différentes opérations possibles (envoi original, dépassement de délai de retransmission, acquittement, et retransmission). Pour détailler tous les comportements possibles avec clarté, on n'a pas respecté sur cette figure les règles décrites au V.5 imposant à l'émetteur d'envoyer les messages un par un, en attendant les acquittements, pendant les phases de retransmissions.

Cette deuxième approche présente un avantage par rapport à la première : on peut maintenant régler la valeur du *timeout* comme on le désire, sans avoir à augmenter la taille de la table. Par contre, on perd une propriété intéressante : lors des retransmissions, le numéro d'index utilisé n'est pas le même (on voit d'ailleurs cela assez bien dans la figure 20). Mais si on intègre les comportements décrits au V.5, on peut garantir la propriété de réutilisation de la même entrée pour le même message retransmis.

Au V.5, nous avons prévu que lorsque l'émetteur constatait qu'un acquittement n'était pas arrivé, il devait passer en mode reprise sur erreur, et n'envoyer les messages qu'un par un, en attendant l'acquittement de chaque message pour passer au suivant. Dans ces conditions, on a plus d'envoi parallèle des messages, et on n'a pas besoin de confier au processus *Watchdog* la détection des acquittement en retard. Un fois qu'un message est retransmis, le processus d'émission attend l'arrivée d'un acquittement ou bien la durée du *timeout*. Si après ce temps aucun acquittement n'est arrivé, il retransmet à nouveau le message.

Comme ce n'est pas le processus *Watchdog* qui gère le timeout des messages retransmis, on peut donc ré-utiliser la même entrée pour toutes les retransmissions d'un même message. Ce mécanisme ne provoquera pas d'interférences avec le fonctionnement du processus *Watchdog* , puisque l'entrée retransmise est située derrière le pointeur W dans la table. W ne pouvant dépasser le pointeur N, on est assuré que *Watchdog* ne viendra pas s'intéresser aux entrées en cours de retransmission gérées entièrement par le processus d'émission.

On peut toujours avoir un problème d'acquittement en retard accusant une mauvaise entrée, si après que le nombre maximum de tentatives d'envoi ait été dépassé, on abandonne l'envoi. Si l'acquittement arrive après cet abandon, on peut acquitter la mauvaise entrée dans la table. Un tel phénomène est peu probable,

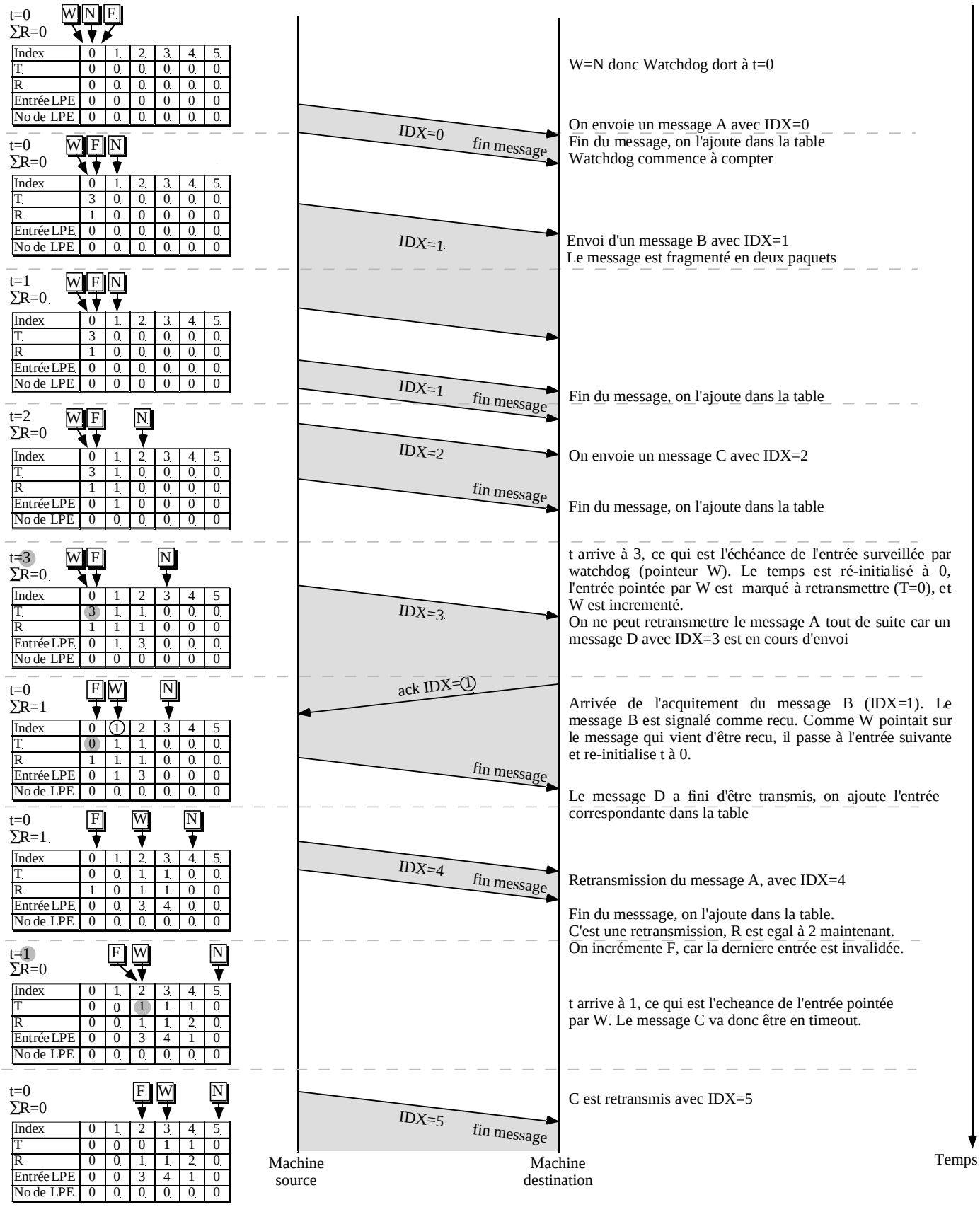
t=0
ΣR=0
W N F
Index 0 1 2 3 4 5
T 0 0 0 0 0 0
R 0 0 0 0 0 0
Entrée LPE 0 0 0 0 0 0
No de LPE 0 0 0 0 0 0
W=N donc Watchdog dort à t=0
IDX=0
fin message
On envoie un message A avec IDX=0
Fin du message, on l'ajoute dans la table
Watchdog commence à compter
t=0
ΣR=0
W F N
Index 0 1 2 3 4 5
T 3 0 0 0 0 0
R 1 0 0 0 0 0
Entrée LPE 0 0 0 0 0 0
No de LPE 0 0 0 0 0 0
IDX=1
Envoi d'un message B avec IDX=1
Le message est fragmenté en deux paquets
t=1
ΣR=0
W F N
Index 0 1 2 3 4 5
T 3 0 0 0 0 0
R 1 0 0 0 0 0
Entrée LPE 0 0 0 0 0 0
No de LPE 0 0 0 0 0 0
IDX=1
fin message
Fin du message, on l'ajoute dans la table
t=2
ΣR=0
W F N
Index 0 1 2 3 4 5
T 3 1 0 0 0 0
R 1 1 0 0 0 0
Entrée LPE 0 1 0 0 0 0
No de LPE 0 0 0 0 0 0
IDX=2
On envoie un message C avec IDX=2
fin message
Fin du message, on l'ajoute dans la table
t=3
ΣR=0
W F N
Index 0 1 2 3 4 5
T 3 1 1 0 0 0
R 1 1 1 0 0 0
Entrée LPE 0 1 3 0 0 0
No de LPE 0 0 0 0 0 0
t arrive à 3, ce qui est l'échéance de l'entrée surveillée par watchdog (pointeur W). Le temps est ré-initialisé à 0, l'entrée pointée par W est marqué à retransmettre (T=0), et W est incrementé.
On ne peut retransmettre le message A tout de suite car un message D avec IDX=3 est en cours d'envoi
IDX=3
t=0
ΣR=1
F W N
Index 0 ① 2 3 4 5
T 0 1 1 0 0 0
R 1 1 1 0 0 0
Entrée LPE 0 1 3 0 0 0
No de LPE 0 0 0 0 0 0
ack IDX=①
Arrivée de l'acquitement du message B (IDX=1). Le message B est signalé comme recu. Comme W pointait sur le message qui vient d'être recu, il passe à l'entrée suivante et re-initialise t à 0.
fin message
Le message D a fini d'être transmis, on ajoute l'entrée correspondante dans la table
t=0
ΣR=1
F W N
Index 0 1 2 3 4 5
T 0 0 1 1 0 0
R 1 0 1 1 0 0
Entrée LPE 0 0 3 4 0 0
No de LPE 0 0 0 0 0 0
IDX=4
fin message
Retransmission du message A, avec IDX=4
Fin du messsage, on l'ajoute dans la table.
C'est une retransmission, R est egal à 2 maintenant.
On incrémente F, car la derniere entrée est invalidée.
t=1
ΣR=0
F W N
Index 0 1 2 3 4 5
T 0 0 1 1 1 0
R 0 0 1 1 2 0
Entrée LPE 0 0 3 4 1 0
No de LPE 0 0 0 0 0 0
t arrive à 1, ce qui est l'echeance de l'entrée pointée par W. Le message C va donc être en timeout.
t=0
ΣR=0
F W N
Index 0 1 2 3 4 5
T 0 0 0 1 1 0
R 0 0 1 1 2 0
Entrée LPE 0 0 3 4 1 0
No de LPE 0 0 0 0 0 0
IDX=5
fin message
C est retransmis avec IDX=5
Machine source
Machine destination
Temps

Fig. 20 : Exemples de manipulation sur la table des messages en transit (avec champs d'échéances). L'échelle de temps n'est pas respectée. On a pris un timeout *de 3 ticks. Pour représenter clairement tous les cas possibles, les comportements décrits au V.5 ne sont pas respectés ici.*

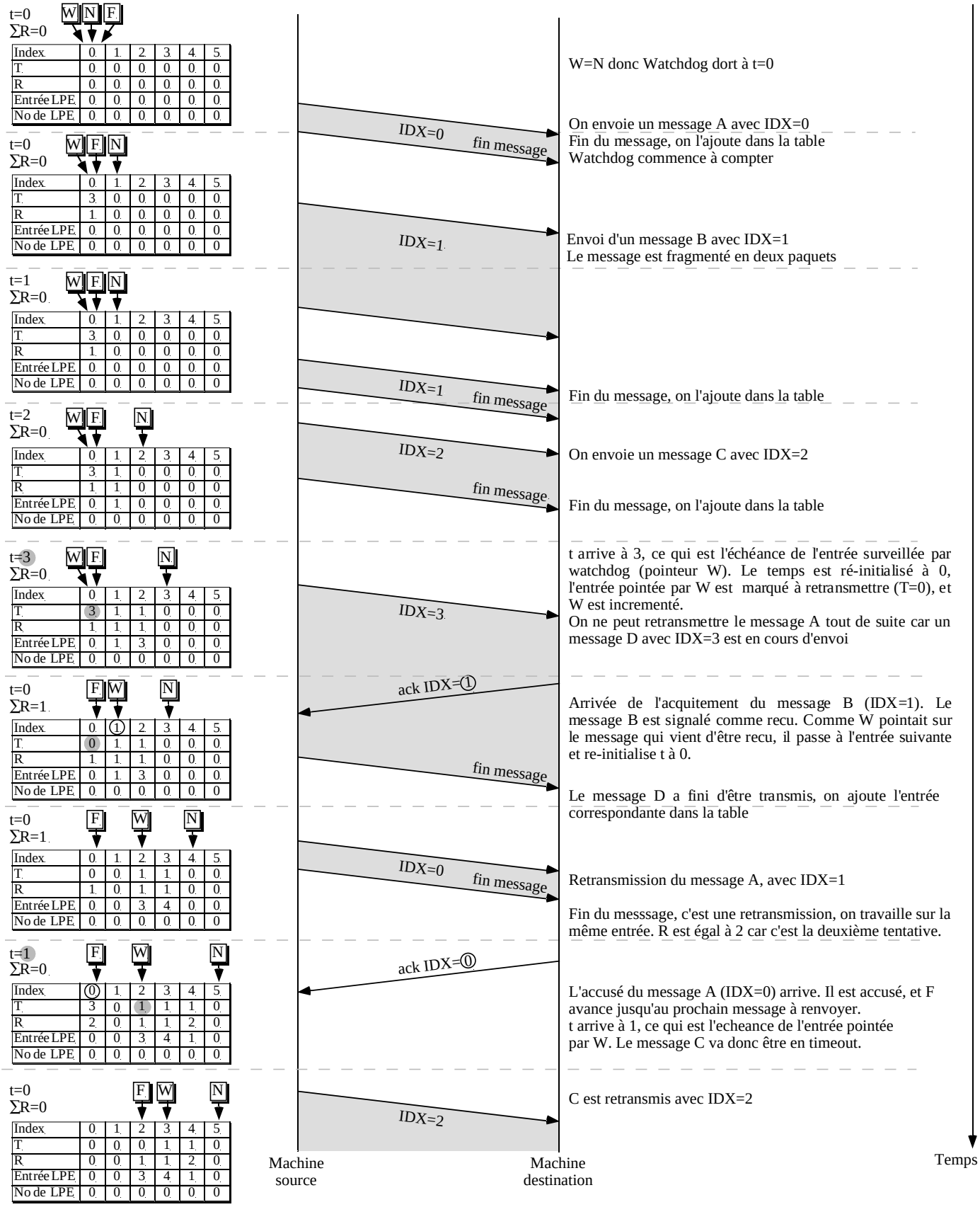
t=0
ΣR=0
W=N donc Watchdog dort à t=0
IDX=0
fin message
On envoie un message A avec IDX=0
Fin du message, on l'ajoute dans la table
Watchdog commence à compter
t=0
ΣR=0
IDX=1
Envoi d'un message B avec IDX=1
Le message est fragmenté en deux paquets
t=1
ΣR=0
IDX=1
fin message
Fin du message, on l'ajoute dans la table
t=2
ΣR=0
IDX=2
On envoie un message C avec IDX=2
fin message
Fin du message, on l'ajoute dans la table
t=3
ΣR=0
t arrive à 3, ce qui est l'échéance de l'entrée surveillée par watchdog (pointeur W). Le temps est ré-initialisé à 0, l'entrée pointée par W est marqué à retransmettre (T=0), et W est incrementé.
On ne peut retransmettre le message A tout de suite car un message D avec IDX=3 est en cours d'envoi
IDX=3
t=0
ΣR=1
ack IDX=①
Arrivée de l'acquitement du message B (IDX=1). Le message B est signalé comme recu. Comme W pointait sur le message qui vient d'être recu, il passe à l'entrée suivante et re-initialise t à 0.
fin message
Le message D a fini d'être transmis, on ajoute l'entrée correspondante dans la table
t=0
ΣR=1
IDX=0
fin message
Retransmission du message A, avec IDX=1
Fin du messsage, c'est une retransmission, on travaille sur la même entrée. R est égal à 2 car c'est la deuxième tentative.
t=1
ΣR=0
ack IDX=⓪
L'accusé du message A (IDX=0) arrive. Il est accusé, et F avance jusqu'au prochain message à renvoyer.
t arrive à 1, ce qui est l'echeance de l'entrée pointée par W. Le message C va donc être en timeout.
t=0
ΣR=0
C est retransmis avec IDX=2
IDX=2
Machine source
Machine destination
Temps

Fig. 21 : Exemples de manipulation sur la table des messages en transit , en intégrant les règles du V.5. Ainsi on arrive à re-envoyer les messages avec le même index.

mais peut exister. La seule façon de l'éviter est de renoncer à la signalisation des échec, et tenter d'émettre le message indéfiniment, quitte à bloquer la machine en cas de câble débranché.

La deuxième version de la table, avec champ d'échéance, nous permet tout ce que permet la première, mais en plus, elle permet de manipuler librement la valeur du *timeout* sans agrandir la table. On va utiliser cette deuxième version, avec les champs d'échéance. La figure 21 donne des exemples d'utilisation de la table, en intégrant les règles données au V.5.

Détaillons maintenant le nombre de champs dans chaque entrée de la table. Dans une entrée de table des messages en transit, nous devons avoir :

• L'échéance de l'entrée. On fixe la largeur de ce champ à 16 bits. Si on compte des ticks de 240ns (ce qui corresponds à la vitesse maximum auquel le matériel peut émettre des messages), un champ d'échéance de 16 bits permet de gérer des délais de retransmission d'au plus 15ms.

• Le nombre de tentatives d'envoi de l'entrée (initialement zéro). On a choisi de prendre ce champ d'une longueur de 4 bits. Ceci donne 16 tentatives d'envoi maximum pour un message. Etant donné que la perte d'un message est un événement rare, c'est une valeur surdimensionnée.

• Le numéro de séquence du premier paquet de la première page du message. On verra dans la partie suivante les raisons qui nous poussent à fixer la taille de ce champ à 16 bits.

• Le numéro de l'entrée de LPE traitée : dans PUT/1, la LPE n'a pas d'autre limite de taille que la mémoire disponible. La présence de ce champ va imposer une taille de LPE maximum dans MPC/2. En prenant ce champ d'une taille de 16 bits, on limite la taille de la LPE à 65536 entrées, ce qui paraît raisonnable.

• Le numéro de LPE : de même que le champ précédent limite la taille de la LPE, ce champ limite le nombre de processus pouvant utiliser simultanément la carte FastHSL. On a choisit de le fixer à 12bits, ce qui donne un nombre maximum de processus pouvant utiliser la carte égal à 4096. étant donné que le nombre de processus fonctionnant sur un système UNIX est nettement moins élevé que cela, ce champ est nettement surdimensionné.

Avec ces valeurs, une entrée occupe 64 bits. On a dit plus haut que l'on souhaitait avoir 62 entrées. Pour donner au numéro d'index dans la table une taille d'un nombre entier d'octet, prenons 256 entrées. Ceci nous fait donc une table de exactement 2ko. Cette valeur est suffisamment faible pour permettre d'embarquer la table dans le FPGA, et donc de gérer totalement matériellement le mécanisme de re-émission des messages perdus. Nous avons donc dans le FPGA bien-sûr un processus d'émission de message, mais aussi un processus de surveillance des messages non acquittés (*Watchdog*).

VI.2-Implémentation du rejet des doublons: les numéros de séquence

Intéressons nous maintenant au mécanisme de rejet des doublons, c'est à dire des messages reçus en double, suite à un accusé de réception perdu ou en retard. Nous avons abordé le problème au V.4, et nous savons que ce comportement est assuré par les numéros de séquences décrits au V.1. Un point important est qu'il ne peut être implémenté que matériellement. En effet, c'est le contrôleur réseau qui dépose les paquets en mémoire. La machine hôte est passive dans cette opération, et n'a aucun contrôle sur le dépôt. Seul le contrôleur réseau peut décider de bloquer un paquet ou de le déposer.

Pour obtenir le système de numéros de séquences décrits dans le V.1, nous devons mettre en place deux structures de données. Sur l'émetteur, chaque paquet émis vers un nœud donné doit porter un numéro de séquence croissant. Il nous faut donc une table de numéros de séquence courant pour chaque nœud destination. Sur le récepteur, il nous faut une table donnant pour chaque nœud source le numéro de séquence attendu.

Ceci nous fait donc deux tables par nœud, chaque nœud pouvant jouer le rôle d'émetteur ou celui de récepteur. Ce sont les tables des séquences, respectivement en transmission et en réception. Ces tables continnent autant d'entrées qu'il y a de nœuds sur le réseau HSL. Chaque entrée de la table en transmission contient le numéro de séquence courant en émission pour le nœud correspondant, et chaque entrée de la table des séquences en réception contient le numéro de séquence attendu en réception pour le nœud correspondant.

L'utilisation des numéros de séquences est beaucoup plus simple que l'utilisation de la table des messages en transit :

Le comportement de l'émetteur est d'utiliser un numéro de séquence croissant pour chaque paquet envoyé pour la première fois vers un nœud donné. Le numéro courant est stocké dans la table des séquences en transmission. Le numéro du premier paquet du message est conservé dans la table des messages en transit. Lors d'une re-émission, ce numéro est récupéré, et le message est renvoyé avec les mêmes numéros de séquences que la première fois. Une fois la re-émission achevée, l'émetteur reprend les séquences là où il les avait laissées.

Pour sa part, le récepteur ne doit accepter (c'est à dire déposer en mémoire) en provenance d'un nœud donné que les paquets ayant les numéros de séquence attendus. Ce mécanisme permet de rejeter les doublons, mais il a un effet de bord important: en cas de paquet perdu, le récepteur va voir arriver beaucoup de paquets n'ayant pas le numéro de séquence attendu avant la ré-expédition du message posant problème. Eventuellement, des paquets appartenant à plusieurs autres messages seront bloqués. Pendant un temps égal au délai de retransmission Δ, le récepteur n'acceptera plus aucune données.

Il nous faut dimensionner les numéros de séquences. Le point important est que pendant un certain intervalle de temps, le même numéro de séquence ne soit pas utilisé pour designer des paquets de messages différents. Ceci est assuré si les numéros de séquences ne bouclent pas pendant toutes les émissions d'un même message. Un message peut être envoyé 16 fois, chaque envoi etant séparé du précédent par un temps de retransmission Δ. Ceci nous fait un temps $16.\Delta = 8ms$ (avec Δ=500µs) pendant lequel des paquets peuvent être envoyés. Ces paquets sont envoyés au rythme maximum de un tous les 240ns, ce qui fait en 8ms un nombre maximum de paquet envoyé égal à 33333. Il faut aussi compter les numéros de séquences utilisés pour le message retransmis. Si on limite la taille des messages à 64Mo, on consomme 16384 numéros de séquence supplémentaires, et donc 65536 numéros de séquences différents suffisent. On peut alors utiliser des numéros de séquences d'une largeur de 16 bits.

Voyons maintenant les profondeurs des tables de séquences. Elles sont déterminées par le nombre de nœuds. Le réseau HSL comporte au maximum 65536 nœuds (pour R-Cube, le numéro du nœud est codé sur 16 bits). Ceci nous fait deux tables des séquences de 65536×16 bits = 256ko. C'est une valeur beaucoup trop importante pour pouvoir faire tenir la table dans le FPGA (celui-ci ne dispose que de 25ko *on chip*), et ce mécanisme doit donc forcement être implémenté dans le matériel. Nous devons donc introduire une contrainte sur le seul paramètre resté libre: le nombre de nœuds maximum dans un réseau HSL. Moyennant de restreindre la taille de la machine MPC à 256 nœuds (ce qui est déjà beaucoup), les tailles des deux tables diminuent à 256×16 bits = 0,5ko chacune. Deux tables de 0,5ko peuvent tenir dans le FPGA.

VI.3-Détection de la réception complète des paquets d'un messages bis

Nous avons expliqué au V.5 la nécessité d'un mécanisme de détection de la réception de tous les paquets d'un message bis, qu'ils aient été déposés ou non. Comme on est assuré du fait qu'un seul message bis peut arriver d'un nœud émetteur à un moment donné, on va pouvoir surveiller les numéros de séquences des messages bis avec une table de numéros de séquences bis attendus, analogue à la table des numéros de séquences attendus décrite au VI.2. Cette table va donc occuper 0,5ko supplémentaires, et peut être gérée matériellement.

Pour détecter que l'on doive initialiser le numéro de séquence bis dans la table, il faut pouvoir repérer le premier paquet d'un message bis. Nous avons donc besoin dans les paquets d'un drapeau indiquant si le paquet est le premier d'un message. Si ce drapeau est présent, le numéro de séquence bis pour le nœud source est initialisé au numéro de séquence du premier paquet du message plus un.

On va ensuite incrémenter le numéro à chaque fois qu'on reçoit un paquet bis avec le numéro de séquence égal au numéro bis attendu. Si on reçoit le dernier paquet du message avec le numéro bis attendu et le drapeau bis, alors on envoie un accusé de réception. La figure 22 donne un exemple d'utilisation de la table.

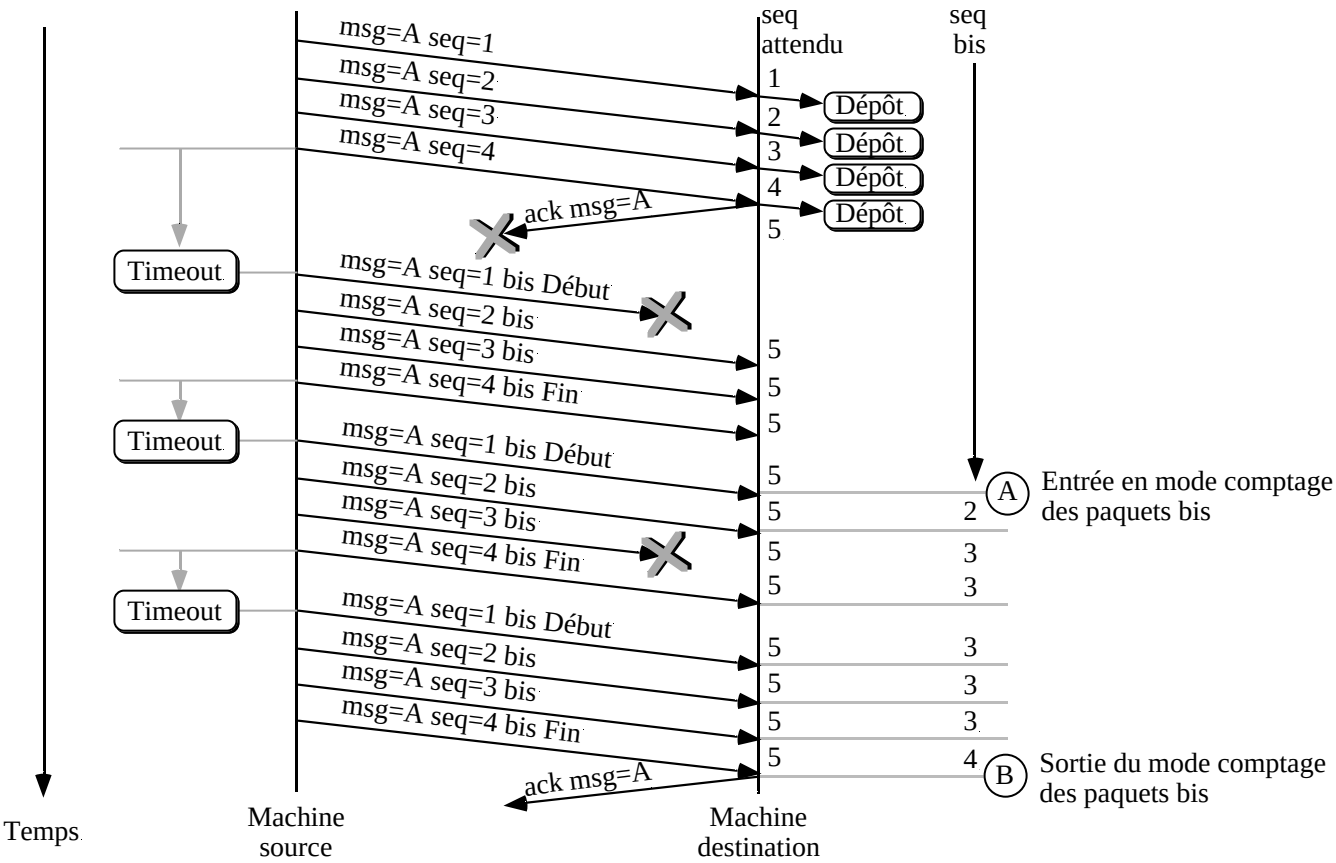

Fig. 22 : Utilisation des numéros de séquence auxiliaire. En A, le récepteur détecte un drapeau bis sur un paquet de début de message. En B, il détecte un paquet bis de fin de message avec un numéro de séquence correspondant au numéro auxiliaire attendu, ceci provoque l'envoi d'un acquittement.

VI.4-Implémentation du mécanisme d'envoi des accusés de réception

Dernier mécanisme à mettre en place: l'envoi des accusés de réception. Ce mécanisme pose un problème particulier: il requiert une forme de communication entre le processus s'occupant de la réception des paquets et le processus s'occupant de l'émission des messages non acquittés. A ce stade de l'étude, nous savons que la retransmission des messages non accusés et le système de numéros de séquences seront gérés dans le matériel. Il s'agit donc de mettre au point la communication entre le processus d'émission et le processus de réception du contrôleur réseau ANI.

Le problème principal qui se présente ici est le fait qu'un nœud ne peut envoyer d'accusés de réception pendant qu'il est en train d'émettre des données. Il n'y a qu'un lien de ANI vers R-Cube. Ceci introduit donc un délai avant l'expédition des accusés. Pour minimiser ce délai, on a décidé de donner aux accusés la priorité la plus haute, et d'expédier tous les accusés en attente entre chaque paquet envoyé.

Ceci signifie qu'il faut mettre en place une file d'attente entre le processus gérant l'envoi des acquittement et le processus d'émission des paquets. Cette file d'attente doit être dimensionnée de telle façon que pendant l'émission d'un paquet de 4ko (nous avons proposé une taille de paquet de 4ko dans le V.7), elle ne déborde pas. Emettre un message de 4ko prend 31,2µs. Pendant ce temps, on peut recevoir un message tous les 240ns, comme on l'a expliqué plus haut, et donc avoir un accusé de plus en attente tous les 240ns. Ceci impose une longueur de file d'attente de 130 places.

Le FPGA peut fournir des files d'attente de 128 ou 256 places. La situation décrite ci dessus etant extrême (dans la réalité, le logiciel n'est pas capable d'envoyer des messages au rythme de un tous les 240ns), on va choisir une longueur de file d'attente de 128 places. Chaque place doit contenir 2 champs:
• Le numéro du nœud destinataire (16 bits)
• L'index du message acquitté (8 bits)

Ces deux champs ont une largeur totale de 24 bits. Le FPGA ne permettant que des largeurs de puissances de 2, donc une place fera 32 bits (avec 8 bits inutilisés). La file d'attente occupe donc 0,5ko, ce qui est raisonnable.

La file d'attente des acquittements est remplie par le processus de réception des paquets. Le processus d'envoi des acquittement la vide en utilisant l'opérateur de construction de paquet (nous verrons tout cela plus en détail dans la partie suivante). Lors de l'usage de cet opérateur, le processus d'envoi des acquittements entre en conflit avec le processus d'émission des messages. Comme il a été précisé précédemment, l'arbitrage va toujours du coté du processus d'acquittement : quand le processus d'émission de messages a envoyé un paquet, il ne peut en envoyer un autre tant que le processus d'envoi des acquittement n'a pas vidé la file d'attente des accusés de réception.

VI.5-Bilan des contraintes imposées par la sécurisation

Nous avons maintenant une idée très claire de ce qui doit être géré par le logiciel et ce qui doit être géré par le matériel. Nous n'avons laissé aucune tâche à accomplir par le logiciel. C'est le FPGA qui prend en charge la totalité des opérations concernant la sécurisation. La mémoire occupée par S/PUT dans le FPGA se répartit comme suit :

Table des messages en transit	2ko
Table des séquences (transmission)	0,5ko
Table des séquences (réception)	0,5ko
Table des séquences (auxiliaire)	0,5ko
File d'attente des acquittements	1ko
Total	3,5ko

Et, par ailleurs, les contraintes imposées:

- Pas de tables de routage adaptatives
- Pas plus de 256 nœuds (avec 65536 nœuds, on aurait besoin de 387ko de mémoire).
- Le découpage des messages en paquets est déterministe.
- L'envoi d'un message est atomique (i.e.: on n'entrelace pas les paquets de deux messages différents).

VI.6-S/PUT-1.0 en bref

Cette partie synthétise les partie précédentes, en présentant les différents processus utilisés par S/PUT-1.0, leur comportement, les structures de données qu'ils manipulent. Le format des paquets est aussi proposé.

VI.6.a-Présentation des structures de données

S/PUT utilise les structures de données suivantes (décrites en langage C pour éviter toute confusion)

```
#define TRANSIT_TABLE_SIZE    256    /* taille de la table des messages en transit */
#define MAX_HSL_NODES         256    /* nombre de nœud maximum sur le réseau */
#define ACK_FIFO_SIZE         128    /* taille de la file d'attente des acquittements */
#define MAX_CHANNELS          4096   /* Nombre de LPE multiples */

/* table des messages en transit */
typedef struct _transit_table_entry {
        u_short         timeout         /* Date d'échéance du retard */
        u_short         lpe_index;      /* Index de l'entrée de LPE de la première page du message */
        u_short         misc            /* Numéro de LPE  (12 bits) + compteur de retransmission (4 bits) */
        u_short         first_seq;      /* Numéro de séquence du premier paquet de la première page */
} transit_table_entry_t;
transit_table_entry_t   transit_table [TRANSIT_TABLE_SIZE];     /* 256 entrées de 64 bits = 2ko */

/* Index travaillant sur la table des messages en transit */
u_char          first_msg;      /* Message le plus ancien (pointeur F) */
u_char          watched_msg; /* Entrée surveillée par le processus Wd (pointeur W) */
u_char          next_msg;       /* première entrée libre dans la table (pointeur N) */

/* table des séquences */
u_short         tx_seq_table    [MAX_HSL_NODES];        /* Séquences en transmission */
u_short         rx_seq_table    [MAX_HSL_NODES];        /* Séquences en réception */
u_short         bis_seq_table   [MAX_HSL_NODES];        /* Séquences bis en réception */
/* 3 table de 256 entrées de 32 bits = 1,5ko */

/* File d'attente des acquittements: 128 entrées de 32bits = 0,5ko*/

/* Total: 4ko de données stokées dans le FPGA */
```

VI.6.b-Présentation des différents processus

Les processus suivants sont nécessaire au fonctionnement de S/PUT:

- Emission de messages (Tx)
- réception de paquets (Rx)
- Surveillance des messages non acquittés (Wd : Watchdog)
- Envoi d'acquittements (Ack)

De plus, on a besoin des opérateurs suivants:

- Constructeur de paquets HSL (Pb: Packet Builder)
- Extracteur de paquets HSL (Pb: Packet Extractor)
- Opérateur de lecture par DMA (DMA-R)
- Opérateur d'écriture par DMA (DMA-W)
- Esclave PCI recevant les ordre d'envoi de messages (PCI)

La figure 23 présente les relations entre ces différentes entités.

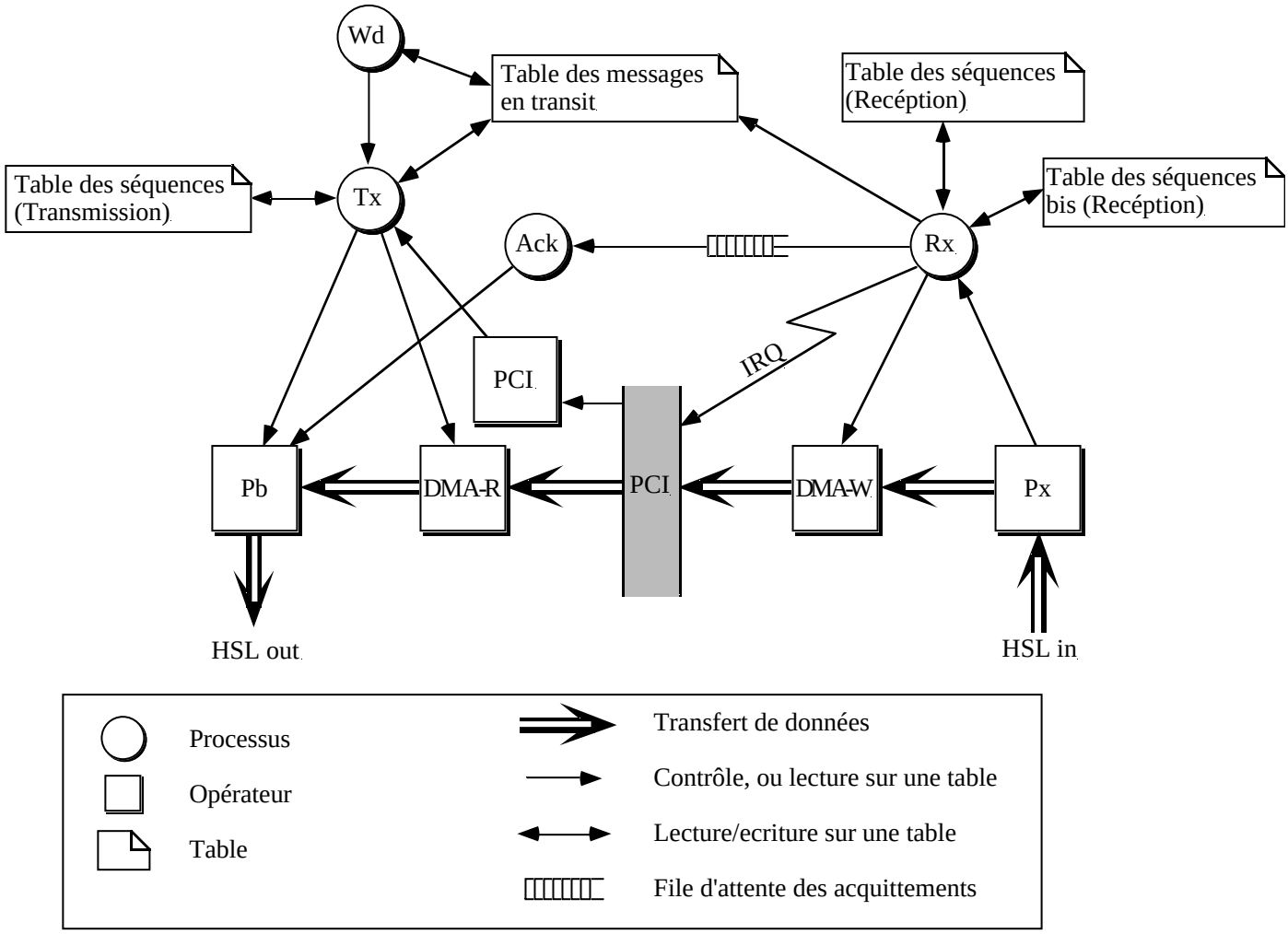

Fig. 23: relations entre les différents processus et opérateurs utilisés par S/PUT. Dans un effort de synthèse, les données importantes manipulées sont aussi représentées

VI.6.c-Opérations sur la table des messages en transit

Comme on l'a dit plus haut, 3 processus manipulent cette table, pour 4 opérations différentes : envoi original ou retransmission pour le processus d'émission (Tx), acquittement pour le processus de réception (Rx), et dépassement du délai de retransmission (*timeout*) pour le processus de surveillance (Wd). Avant de détailler les opérations, rappelons un certain nombre de notations:

- T Le champ échéance d'une entrée de la table
- R Le nombre de tentatives de renvoi d'une entrée de la table
- t Le temps courant manipulé par Wd. Il est exprimé en ticks de $\Delta t = 240ns$
- Δ La délai de retransmission (*timeout*)
- n Le nombre d'entrées dans la table
- $\sum T$ La somme des échéances des entrées de la table
- $\sum R$ Le nombre d'entrées en instance de renvoi, c'est a dire ayant le champ $R \neq 0$
- N Le pointeur de première entrée libre
- F Le pointeur du plus ancien message
- W Le pointeur sur le prochain message en *timeout*

Rappelons les conditions importantes énoncées au VI.1.b.

- La table est vide si N = F
- La table est non pleine si $N+1 \neq F$ [n]
- Un élément est valide (il décrit un message en transit) si $R \neq 0$, sinon il est invalide.
- Un élément décrit un message en instance de retransmission si il a T =0 et $R \neq 0$.
- Les éléments valides sont entre F (inclus) et N (exclu).
- Les éléments en instance de retransmission sont entre F (inclu) et W (exclu).
- Les éléments valides en attente d'acquittement sont entre W (inclu) et N (exclu).

Et voici le détail de chaque opération sur la table:

```
Emission originale (Tx)
    Si (table non pleine)
        (envoi du message)
        table[N].R = 1
        table[N].T = t + Δ - ∑T
        ∑T = Δ - t
        N = N + 1

Dépassement du délais de retransmission (Wd)
    faire
        si (table[W].T > t) et (table[W].R ≠ 0)
            t = 0
            ∑T = ∑T - table[W].T
            table[W].T = 0
            ∑R = ∑R + 1
        tant que (table[W].R = 0) et (W ≠ N)
            W = W + 1
        attendre Δt
        tant que (W = N)
            attendre Δt
        t = t + 1
```

```
Accusé de réception du message IDX (Rx)
    si (table[IDX].R ≠ 0) et (table[IDX].T ≠ 0)
        table[IDX].R = 0
        (signalisation de la complétion de l'envoi)
        Rp=table[IDX].T
        i = IDX
        faire
            i=i+1
        tant que (i ≠ N) et (table[i].R = 0)
        si (i ≠ N)
            table[i].T = table[i].T + Rp
        sinon
            ∑T = ∑T - Rp
        si (IDX = F)
            F = i

Retransmission (Tx)
    tant que (∑R ≠ 0)
        si (table[F].R ≠ 0) et (table[F].T = 0)
            tant que (table[i].R ≠ 0) et (table[i].R ≠ max_retry)
                (envoi du message)
                table[F].R = table[F].R + 1
                table[F].T = Δ
                tant que (table[i].R ≠ 0) et (table[i].T ≠ 0)    # Attente timeout ou ack.
                    attendre Δt
                    table[F].T = table[F].T - 1
            si (table[i].R = max_retry)
                table[i].R = 0
                (signaler l'échec)
            ∑R = ∑R - 1
        F = F + 1
```

VI.6.d-Comportement du processus d'émission

Le processus d'émission (Tx) doit gérer deux choses différentes: l'émission originale d'un message, et la retransmission d'un message. Cette seconde opération est prioritaire sur la première. Tant qu'il y a des messages en retard, le processus Tx doit les ré-émettre avant de pouvoir envoyer un nouveau message. C'est le processus Wd qui indique au processus Tx qu'un message doit être retransmis.

La communication entre Wd et Tx se fait par la variable $\sum R$, qui indique le nombre de messages en retard à traiter. Wd l'incrémente quand il découvre un message en retard, et Tx la décrémente lorsqu'il a renvoyé un message. Tx ne peut donc pas faire des envois originaux de messages tant que $\sum R$ n'est pas nul.

L'opérateur PCI est un esclave sur le bus PCI. C'est lui qui demande à Tx l'envoi de nouveaux messages. Tx prend en charge les émissions commandées par PCI si $\sum R = 0$.

Tx a donc 3 états de fonctionnements: *Idle*, *Insert* et *Replay*, correspondant respectivement à l'inactivité, l'envoi original de message, et la retransmission de message. La figure 24 décrit l'automate à états, avec les conditions pour passer d'un état à l'autre.

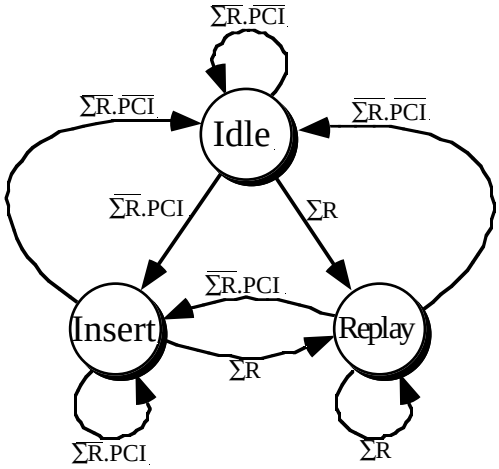

Fig. 24 : Automate d'états de Tx. $\sum R$ indique des messages à retransmettre (état Replay*), et PCI indique des envois originaux de messages à faire (état* Insert*).*

La figure 25 décrit l'opération commune aux états *Insert* et *Replay*, à savoir l'émission d'un message. La figure 26 décrit l'état *Insert* et l'état *Replay*.

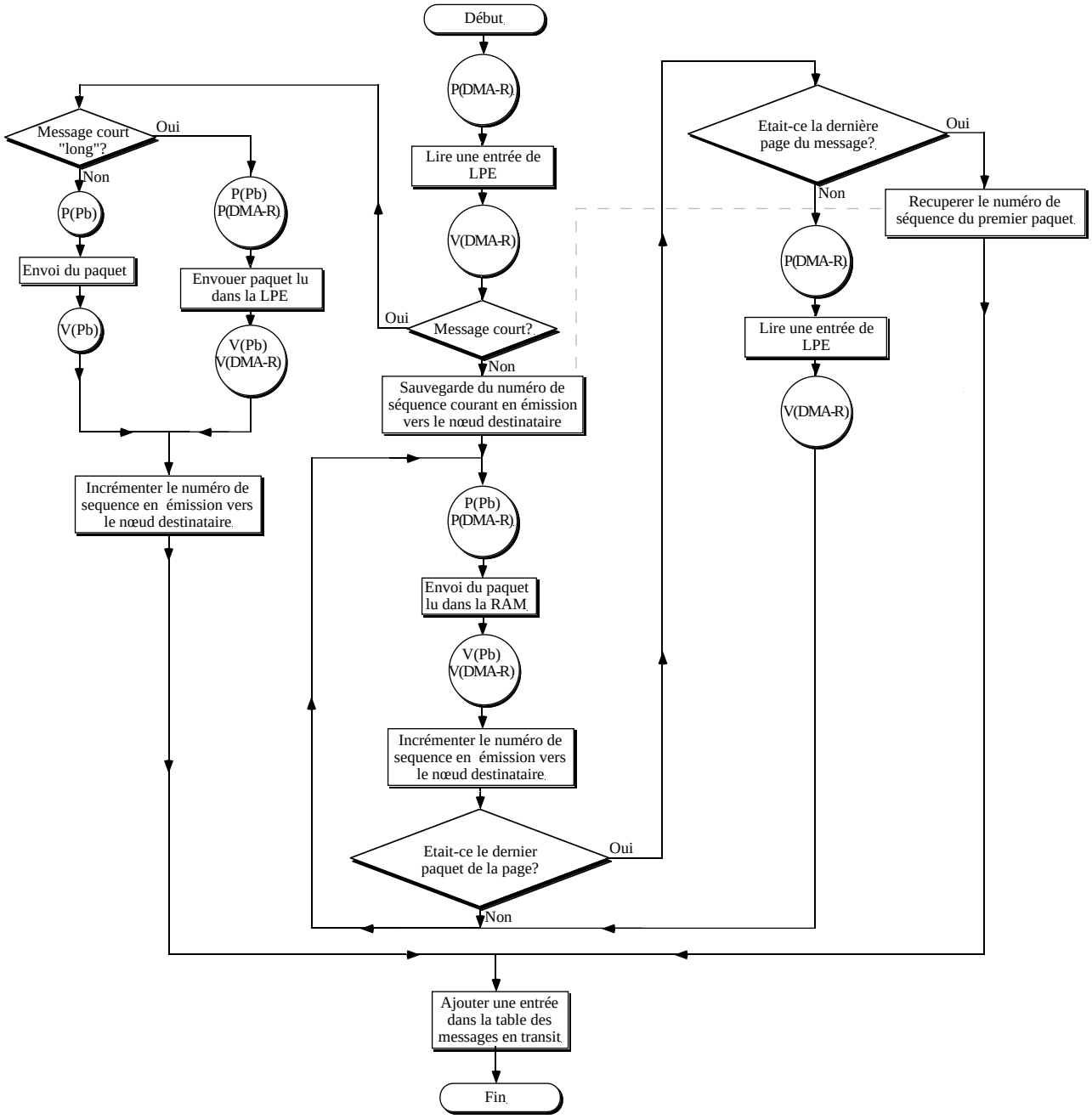

Fig. 25 : Partie du processus d'émission qui réalise l'opération d'envoi d'un message. Cette parti e est utilisée par les états Insert et Replay.

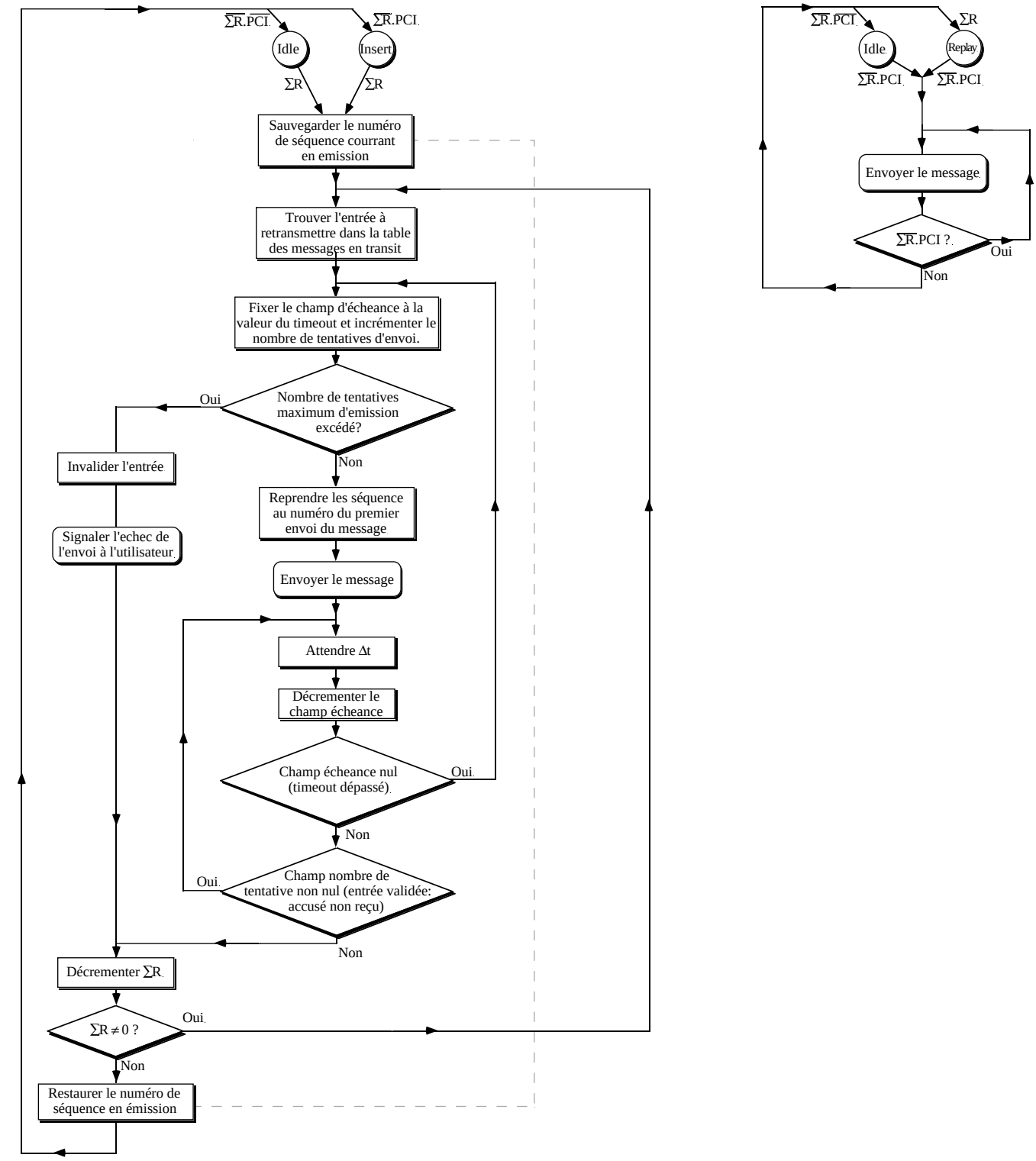

Fig. 26 : les états Replay *(à gauche) et* Insert *(à droite) du processus Tx. Ces deux états partagent l'opération d'envoi d'un message, qui est décrite dans la figure 25.*

VI.6.e-Comportement du processus de surveillance des messages non acquittés

Le processus Wd est chargé de surveiller les messages dont l'acquittement est en retard. Wd surveille une entrée de la table des messages en transit, pointée par W. Lorsque le délai de retransmission de cette entrée arrive à échéance, l'entrée est marquée à retransmettre (champ échéance nul), et le processus Tx est notifié de la présence de message à retransmettre par l'incrémentation du compteur ∑R.

La figure 27 donne le comportement du processus Wd.

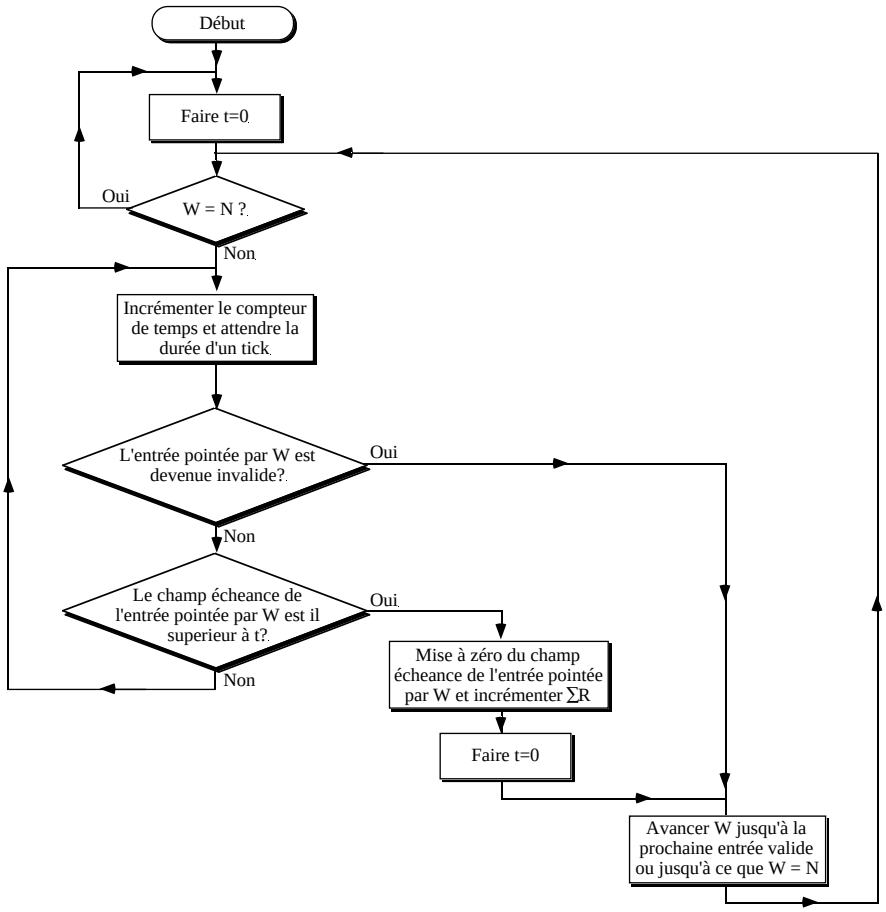
Début
Faire t=0
W = N ?
Oui
Non
Incrémenter le compteur de temps et attendre la durée d'un tick
L'entrée pointée par W est devenue invalide?
Oui
Non
Le champ écheance de l'entrée pointée par W est il superieur à t?
Oui
Non
Mise à zéro du champ écheance de l'entrée pointée par W et incrémenter ∑R
Faire t=0
Avancer W jusqu'à la prochaine entrée valide ou jusqu'à ce que W = N

Fig. 27 : Comportement du processus Wd. N et W sont les pointeurs dans le table des messages en transit, respectivement sur l'entrée surveillée et sur la première entrée libre.

VI.6.f-Comportement du processus de réception

Le processus de réception (Rx) récupère les paquets extraits par l'opérateur Px. Son comportement varie ensuite suivant le type du paquet. Les acquittements donnent lieu à une opération sur la table des messages en transit, et les paquets appartenant à des messages (courts ou normaux) sont déposés en mémoire si le numéro de séquence est correct. La figure 28 détaille le comportement du processus Rx.

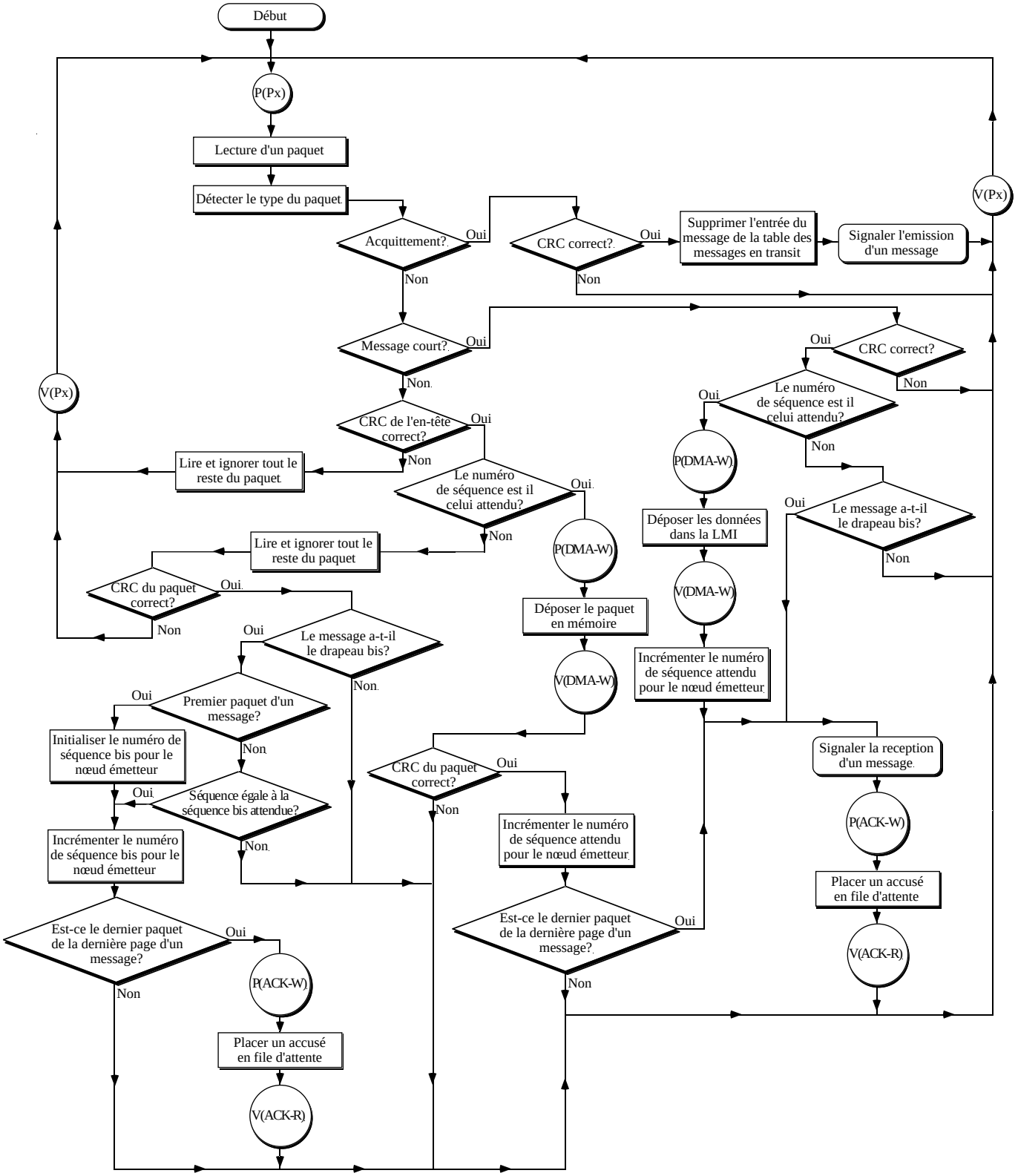

Fig. 28 : comportement du processus Rx

VI.6.g-Comportement du processus d'envoi des acquittements

Le processus d'envoi des acquittements (ack) lit les accusés de réception à envoyer dans la file d'attente des accusés de réception. Lorsqu'un accusé est prêt à être envoyé, Ack est prioritaire sur Tx pour utiliser le constructeur de paquet (Pb) et envoyer ses acquittements. Tx finit donc d'envoyer le paquet qu'il traitait, puis Ack envoie ses acquittements. Une fois que Ack n'a plus d'acquittements en attente, Tx peut reprendre son travail.

La synchronisation entre Rx et Ack se fait par deux sémaphores ACK-R et ACK-W, utilisés respectivement pour la lecture et l'écriture dans la file.

La figure 29 donne le comportement du processus Ack.

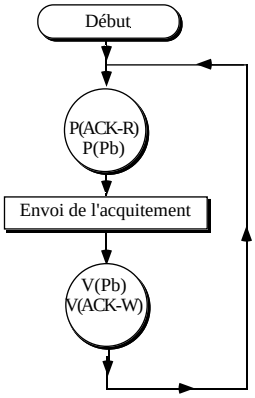

Fig. 29 : comportement du processus Ack

VI.6.h-Formats des paquets

La figure 30 propose les formats des 3 paquets différents utilisés par S/PUT. Il faut noter que dans la proposition d'implémentation, on a limité le nombre de nœuds à 256, pour économiser de la mémoire dans le FPGA, et dans les paquets, les adresses de nœuds sont sur 16 bits. La raison en est simple : si dans le futur on ajoute de la mémoire au FPGA, on pourra stoker la table de 256ko nécessaire à la gestion de 65536 nœuds. On prévoit donc des numéros de nœuds sur 16 bits partout dans le circuit, et on se contente de limiter aujourd'hui à 8 bits dans les structures de données utilisées. Ainsi la transition à 16 bits sera triviale (il suffira d'augmenter la taille de la table).

Le dimensionnement des paquets d'accusés de réception réponds à des nécessités simples: on a besoin des champs System Flags (4 bits), Index (8 bits), les numéros de nœuds source et destination (16 bits chacun). Ceci fait un total de 28 bits. A cela il faut ajouter 8 bits pour le caractère de fin de paquet End of Packet. On arrive à 36 bits. On souhaite avoir un paquet long d'un multiple de 32 bits. Si on plaçait un CRC avec la même disposition que dans les paquets classiques, on devrait compter au moins 128 bits au total. On préfère donc se passer de CRC et doubler toutes les données, pour les protéger des erreurs. Le paquet d'acquittement fait donc 12 octets.

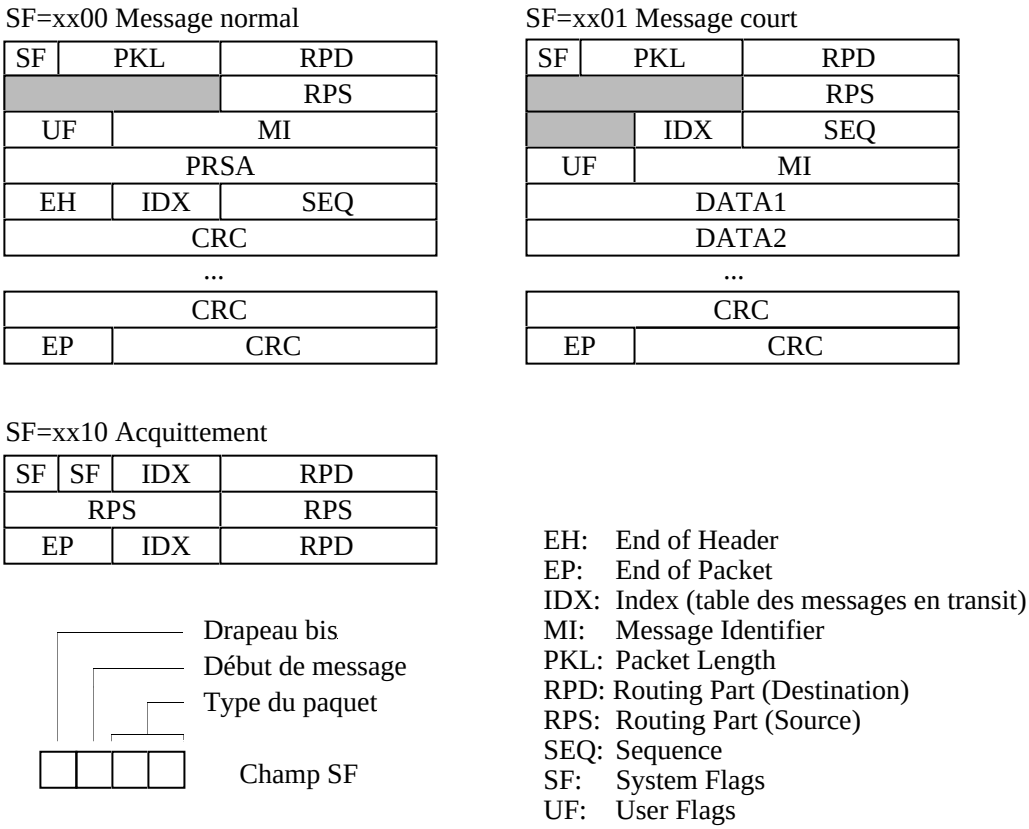

Fig. 30 : format des paquets utilisés par S/PUT. Le format des acquittements est un peu particulier : comme on n'a pas la place de placer un CRC, on envoie tous les champs en double.

VII-Performances

Pour finir, examinons les performances obtenues avec S/PUT-1.0. pour cela, on va calculer les débits et latences moyens obtenus avec le protocole de correction d'erreur. On va supposer pour cela qu'on a pas de doubles fautes. On va envisager le pire des cas, qui est celui d'un paquet perdu : pour un accusé perdu, on doit renvoyer seulement le message dont l'accusé a été perdu, alors que pour un paquet perdu, il faut renvoyer autant de messages qu'il y a de places dans la table des messages en transit.

VII.a-Latence

Pour mesurer la latence, on s'intéresse à l'envoi en continu de petits paquets, disons des messages courts. En temps normal, la latence d'un tel message est d'environ 5µs. En cas de perte de paquet, il faut retransmettre autant de messages qu'on peut en envoyer avant de constater que le message perdu n'est pas acquitté, c'est à dire autant de messages qu'on peut en envoyer pendant la durée du *timeout*.. Ces messages sont retardés du temps d'envoi normal (4µs), plus le délai de retransmission (Δ = 500µs), plus le temps d'arrivé de l'accusé de réception (Tack = 1µs) (puisqu'en mode retransmission, on envoie les messages un par un). Ceci fait un total d'environ 505µs.

Si on suppose qu'on peut envoyer les messages courts au rythme de un tous les Tm = 8µs (le logiciel ne peut en envoyer plus vite), avant d'avoir un *timeout* (500µs), on peut envoyer Δ / Tm = 63 messages. Nous avons donc 63 messages qui subiront une latence anormalement élevée à 505µs à cause de la faute. Le MTBF est de deux minutes. Pendant ce temps, au mieux, on peut envoyer 30 millions de messages, au rythme de un message tous les 8µs. Ces messages ont une latence normale de 5µs. Si on calcule la latence moyenne, on obtient un accroissement de 0,38ns par rapport à la situation sans erreur, où la latence était de 5µs.. C'est un chiffre satisfaisant.

VII.b-Débit

Pour calculer le débit, on va supposer l'envoi en continu de messages de taille identique. Dans le cas d'un message perdu, on va avoir une période où tous les messages seront rejetés, et où aucune communication ne sera possible. Cette période dure Ti = 505µs, comme on l'a expliqué dans le paragraphe concernant la latence. Ensuite vient la période où l'on retransmet les messages non acquittés, un par un. Si les messages ont une taille N, on a mis Tm = 30.(12+E(N/4) nanosecondes pour les envoyer. A ceci il faut ajouter un temps Tack d'une microseconde pour l'arrivée de l'acquittement.

Pendant la durée du délai de retransmission, Δ = 500µs, on a pu envoyer Δ / Tm messages, qui doivent être retransmis, au rythme de un message tous les Tm + Tack. Cette opération prend donc :

$$[\Delta / Tm] \times [Tm + Tack] .$$

On a donc un temps $Te = Ti + [\Delta / Tm] \times [Tm + Tack]$ pendant lequel sont envoyés Tm messages de N octets, soit un débit $De = Tm.N / (Ti + [\Delta / Tm] \times [Tm + Tack])$.

Avec un MTBF de deux minutes, on a un débit maximum pendant deux minutes, puis un temps Te pendant lequel le débit est égal à D. Si on prend des messages assez grands (par exemple 64ko), le débit sans faute est celui du bus PCI, 132Mo/s. Si on fait la moyenne, on obtient une perte de débit par rapport au cas sans faute égal à 1,1ko/s, ce qui est aussi un chiffre satisfaisant.

Ces calculs, aussi bien pour le débit que pour la latence, sont assez grossiers. Pour obtenir des résultats plus satisfaisants, une étude probabiliste serait nécessaire. On pourrait alors proposer un modèle d'arrivée des messages correspondant plus à la réalité qu'un flux ininterrompu de messages de tailles toujours identique. L'intérêt des ces calculs rapide de latence et de débit est de donner une idée de l'ordre de grandeur de la dégradation des performances due à la récupération sur erreur.

VII.c-MTBF

On peut s'intéresser au MTBF. Le but de cette partie n'est pas de calculer sa nouvelle valeur, due à l'utilisation de S/PUT, mais de donner une idée des erreurs que ne corrigent pas S/PUT, et qui devraient être prises en compte pour calculer le MTBF de la machine MPC utilisant S/PUT.

On aura des fautes dans les cas suivants:

• Une erreur se glisse dans un paquet sans qu'elle soit détectée par le CRC. On a un CRC sur 56 bits sur les paquets des messages et tous les champs sont en doublés dans les accusés de réception.

• Si on a choisi de signaler un échec de transmission après un certain nombre de tentatives, alors on peut avoir une faute si un accusé arrive alors que l'émetteur a abandonné l'envoi. Si on envoie un message 16 fois, cela signifie que l'accusé a mis $16.\Delta = 8ms$ a traverser le réseau, ou bien qu'il a subit une hexadécuple faute.

Tous les autres problèmes sont à priori couverts par S/PUT. Une étude probabiliste permettrait de calculer le nouveau MTBF, qui devrait avoir une valeur très grande devant les deux minutes actuelles.

VIII-Conclusions

Au cours de ce stage, j'ai tenté de concevoir un système de sécurisation de la primitive de communication bas niveau de la machine MPC. Le but était d'augmenter le MTBF, initialement de deux minutes, à une valeur plus acceptable, idéalement l'infini. Cette tâche passait par l'élaboration d'un protocole, et par la réflexion sur la répartition des tâches entre le matériel et le logiciel. Après étude approfondie, il s'avère que la sécurisation est possible, au prix d'une modification de la sémantique de la primitive de communication bas niveau. Avec PUT, on signale sur l'émetteur la fin de l'envoi des pages. Avec S/PUT, on signale sur l'émetteur la réception des messages par le nœud destinataire.

La conclusion la plus importante de cette étude est sans doute que la totalité du mécanisme de sécurisation peut être implémenté matériellement à un coût raisonnable. Ceci nous permet de ne pas trop dégrader les performances, et nous assure la portabilité du mécanisme de sécurisation d'un système d'exploitation à l'autre. Il est aussi intéressant de signaler que l'étude du problème de sécurisation a souvent donné lieu à des problèmes annexes, n'ayant plus de rapport direct avec le problème de sécurisation. Ces problèmes sont évoqués en annexe du présent rapport.

Quelques points importants ne sont pas abordés dans le présent document. Il s'agit par exemple de la preuve formelle de la validité du protocole proposé pour S/PUT, c'est à dire la vérification qu'il produise une signalisation correcte où un message est signalé, sur l'émetteur et sur le récepteur, si et seulement si il a été intégralement déposé en mémoire. L'étude probabiliste menant à l'expression du MTBF avec S/PUT fait aussi partie des points non abordés. Avec l'implémentation de S/PUT dans ANI, ces deux problèmes restent à traiter.

Références

[AF] *Conception et réalisation d'un noyau de communication bâti sur la primitive d'écriture distante de la machine parallèle MPC*. Alexandre Fenyö (Thèse de doctorat à soutenir octobre 2000)

[BSD] *Design and implementation of the 4.4BSD operating system*
McKusick Bostic, Karels, et Quarterman, Addison-Wesley, Reading, MA, 1996

[DDC] *PCIDDC specification*
http://mpc.lip6.fr/pciddc/spec_pciddc.ps.gz

[ED] *Finalisation du portage de PUT et portage de MPI sur la machine MPC/Linux.*
Emmanuel Dreyfus, rapport de stage ingénieur, juin 2000

[EMU] *Emulateur Logiciel MPC/1 Manuel d'Installation*
http://mpc.lip6.Fr/docemu/docemu.ps.gz

[R3] *R-Cube Specification*
http://mpc.lip6.fr/rcube/r3.v1.7.ps.gz

[FW] Communication orale de Franck Wajsbürt sur les temps de transfert sur le réseau HSL.

[JLD] *Conception et réalisation d'un contrôleur de réseau programmable pour machine parallèle de type grappe de PC*. Jean-Lou Desbarbieux, thèse de doctorat, juin 2000

[MPC] *Interface Logicielle MPC/1 Manuel du Programmeur*
http://mpc.lip6.fr/docutil/docutil.ps.gz

[MPI] *The MPC parallel computer. Hardware low level protocol and perofmances*
Amal Zerrouki, Olivier Glück, Alain Greiner, Franck Wajsbürt et Alexandre Fenyö.
(soumis pour publication)

Annexe : des idées en vrac

Comme on l'a vu au II.6, il y a un certain nombre de fonctionnalités qui n'existent pas dans MPC/1 et qu'on souhaiterait voir dans MPC/2. Celles ci n'ayant pas toujours de rapport avec la sécurisation des communications, elles n'ont pas été évoquées en détail dans le présent rapport. Dans cette annexe, on propose quelques pistes pour ces fonctionnalités

Des messages courts "longs"

Cette fonctionnalité a été examinée dans le détail et est même présente dans les diagrammes détaillant le fonctionnement des processus. Il s'agit d'utiliser la LPE et la LMI comme tampon respectivement d'émission et de réception. Lorsqu'une entrée de type message court est lue, le champ Length, qui indique habituellement la longueur de la page à transférer, indique la longueur du message court. S'il est inférieur à 8, il s'agit d'un message court au sens de MPC/1, sinon, on va transférer les entrées suivantes de la LPE et le déposer dans la LMI. On peut ainsi avoir des messages courts de plusieurs dizaines d'octets si on le souhaite.

Des messages courts encapsulés dans les messages

Une critique vive qui existait était le fait que le MI était trop court. Nous avons proposé des mesures pour restituer la totalité du MI à l'utilisateur, mais il reste qu'il serait agréable de disposer de MI de l'ordre de la dizaine d'octet. Une solution est de permettre la présence de messages courts encapsulés à l'intérieur des messages standards. Actuellement, une entrée de LPE de message court perdue au milieu des entrées de LPE de message normal est une erreur. On pourrait très bien admettre qu'on déplace un message court entre deux pages, et que le tout forme un message. Dans ce cas, on peut très bien commencer un message par une entrée de message court, et ainsi disposer d'octets supplémentaires transférés de la LPE à la LMI pour concevoir des protocoles de haut niveau.

Des envois de messages entrelacés

On a décidé dans le présent document que l'envoi d'un message était atomique : il est impossible d'envoyer deux messages à la fois, en entrelaçant les pages ou les paquets. Ceci est dû au fait que matériellement, c'est beaucoup plus facile à gérer. Etant donné que la taille des messages n'est actuellement pas limitée, on peut se retrouver dans des situations où un processus va accaparer toutes les ressources en envoyant un message trop gros, détériorant ainsi les performances des communications des autres processus. L'entrelacement des messages est donc à étudier.

Signal d'avancée dans la LPE

Avec PCIDDC, on signale au contrôleur réseau qu'on a ajouté une entrée de LPE en déplaçant un pointeur par un accès en configuration. PCIDDC va ensuite lire l'entrée en mémoire par DMA.

Une idée d'optimisation serait de donner directement l'entrée de LPE à ANI par accès mémoire. Soit ANI n'a pas d'entrées en retard, et il peut alors traiter immédiatement l'entrée, soit il y a des entrées en retard. Dans ce dernier cas, l'accès mémoire provoque simplement une avancée du pointeur, et ANI viendra lire par DMA le descripteur de LPE en temps utile.

Par cette méthode, on peut gagner en latence, en évitant un accès PCI quand ANI n'est pas en train de traiter d'entrée de LPE.